HOCHSCHULE DER MEDIEN

Evaluation von Frontend – Optimierungstechniken für das HTTP/2 – Protokoll unter besonderer Beachtung von Server Push

Masterthesis

zur Erlangung des akademischen Grades Master of Arts
im Studiengang Elektronische Medien
im Schwerpunkt Audiovisuelle Medien
an der Hochschule der Medien Stuttgart

vorgelegt von

Iuliia Poberezhnaia

Erstprüfer: Prof. Walter Kriha

Zweitprüfer: Prof. Dipl.-Ing. Uwe Schulz

Bearbeitungszeitraum: 10. Juli 2016 bis 09. November 2016

Stuttgart, November 2016

Hiermit versichere ich, Iuliia Poberezhnaia, ehrenwörtlich, dass ich die vorliegende Masterarbeit mit dem Titel: „Evaluation von Frontend – Optimierungstechniken für das HTTP2 – Protokoll unter besonderer Beachtung von Server Push“ selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen § 19 Abs. 2 Master-SPO der HdM einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

Danksagung

Diese Arbeit wäre nicht ohne die Hilfe verschiedener Personen möglich gewesen. An dieser Stelle möchte ich mich zuerst bei Herrn Walter Kriha bedanken, der mich bei der Ideenfindung unterstützte, sowie für neue Ansätze sorgte und die Arbeit so in die richtige Richtung lenkte. Außerdem möchte ich mich bei Herrn Prof. Dipl.-Ing. Uwe Schulz bedanken, der mir in der Vorlesung relevante praktische Kenntnisse beigebracht hat, ohne die diese Arbeit nicht entstanden wäre. Außerdem bin ich für neue Anregungen und Denkanstöße sehr dankbar.

Daneben gilt mein besonderer Dank Herrn Robin Schulte, der mir sowohl in der Vorlesung als auch im persönlichen Gespräch viel zum Thema Linux und Serverkonfiguration beigebracht hat. Herrn Jakob Schröter danke ich für die Unterstützung beim Thema Webperformance – Optimierung, hilfreiche Tipps, mögliche Ansätze und Ideen zum Aufbau der Arbeit.

Für die Korrektur der Arbeit und die persönliche Unterstützung gebührt mein herzlicher Dank Herrn David Kirschenheuter.

Mein abschließender Dank gilt der Fakultät Elektronische Medien und der Hochschule der Medien für die Ermöglichung eines außergewöhnlichen Masterstudiums!

Inhaltsverzeichnis

1. Einleitung 1
1.1. Motivation und Problemstellung 1
1.2. Zielsetzung 3
1.3. Aufbau der Arbeit 4
2. Der kritische Rendering – Pfad und seine wichtigsten Komponenten 5
2.1. Aufbau des Document Object Model 6
2.2. Aufbau des CSS Object Model 7
2.3. Render – Baumstruktur 7
2.4. „Navigation Timing API" 10
2.5. „Resource Timing API" 13
2.6. Zwischenfazit 15
3. Die Probleme des HTTP/1.1 – Protokolls, das HTTP/2 – Protokoll und deren Optimierungsmöglichkeiten 16
3.1. HTTP/1.1 – Optimierungstechniken, Stand bisher 16
3.1.1. Nutzung von mehreren TCP – Verbindungen 18
3.1.2. Aufteilung von Ressourcen auf mehrere Domains („Domain Sharding") 20
3.1.3. Zusammenfassung von Ressourcen und „Spriting" von Bildern 21
3.1.4. Ergänzung des Codes in der HTML Datei („Ressource Inlining") 22
3.2. „Evergreen" Frontend – Optimierungsmöglichkeiten 23
3.2.1. Browser Caching für statische Ressourcen 24
3.2.2. Datenkompression während des Transports 25
3.2.3. Reduzierung der Datenmenge der Ressourcen 25
3.2.4. Bildoptimierung 26
3.3. Vorstellung des HTTP/2 – Protokolls 28
3.3.1. Wichtigste Bestandteile des HTTP/2 – Protokolls 30
3.3.2. Request und Response Multiplexierung 34
3.3.3. Stream Priorisierung 35
3.3.4. Header Kompression 37
3.3.5. Eine TCP – Verbindung pro Domain 38
3.3.6. Flow Control 40
3.3.7. Server Push 40
3.3.8. Verbindungsaufbau unter dem HTTP/2 – Protokoll 42

3.4. Mögliche Optimierungstechniken für das HTTP/2 – Protokoll 44
4. Vorbereitung der Tests und zur Testevaluation 48
4.1. Vorbereitung der Tests 48
4.2. Vorbereitung zur Testevaluation 54
5. Tests und Testevaluation 63
5.1. Wie funktioniert „Server Push“? 63
5.2. Zwischenfazit 67
5.3. Serverseitige Priorisierung von per „Server Push“ übergebenen Ressourcen 69
5.3.1. Test 1: Priorisierung von per „Server Push“ übergebenen CSS Dateien und Schriften 69
5.3.2. Test 2: Priorisierung von per „Server Push“ übergebenen CSS Dateien und Bildern 71
5.4. Zwischenfazit 72
5.5. Untersuchungen zum „Server Push“ 73
5.5.1. Test 1: „Server Push“ – Einsatz für kritische CSS Ressourcen 74
5.5.2. Zwischenfazit 80
5.5.3. Test 2: „Server Push“ – Einsatz für nicht kritische JavaScript Ressourcen 80
5.5.4. Test 3: „Server Push“ – Einsatz für kritische CSS Ressourcen und nicht kritische Ressourcen (Schriften) 81
5.5.5. Zwischenfazit 88
5.5.6. Test 4: „Server Push“ – Einsatz für kritische CSS Ressourcen und nicht kritische Ressourcen (Bilder) 89
5.5.7. Zwischenfazit 94
5.6. Zwischenfazit 94
5.7. Vergleich des HTTP/1.1 mit dem HTTP/2 – Protokoll 95
5.7.1. Test 1: Untersuchung der Startseite der Webapplikation unter dem HTTP/1.1- und dem HTTP/2 – Protokoll 99
5.7.2. Zwischenfazit 104
5.7.3. Test 2: Untersuchung der Dozenten-Seite unter den Protokollen HTTP/1.1- und HTTP/2. 105
5.7.4. Zwischenfazit 111
5.8. Spielt die Ressourcenaufteilung eine Rolle für das HTTP/2 – Protokoll? 111
5.8.1. Zwischenfazit 114
5.9. Wie funktioniert das HTTP/2 – Protokoll in anderen Browsern? 117
5.9.1. Test 1: Gibt es deutliche Unterschiede in der Bearbeitungsart des neuen Protokolls zwischen den drei populärsten Desktop-Browsern? 117

5.9.2. Zwischenfazit 119
5.9.3. Test2: Wie wird der Browser „Google Chrome“ per „Server Push“ übergebene Ressourcen interpretieren? 120
5.9.4. Zwischenfazit 122
5.10. Angetroffene Schwierigkeiten während der Testdurchführung und der Testevaluation 122
6. Fazit und Ausblick 125
7. Literaturverzeichnis 128
8. Kurzfassung / Abstract 138
9. Anhang 139
9.1. Upgrade HTTP/2 139
9.2. Browserstatistik 141

1. Einleitung

1.1. Motivation und Problemstellung

In den letzten Jahren sind Internetverbindungen deutlich schneller geworden. Allerdings ist die Ladezeit einer Webseite im Vergleich zu früheren Jahren fast gleich geblieben. Der Grund dafür ist, dass die verfügbare Bandbreite im Vergleich zur Menge der Daten einer Webseite fast linear wächst. In den ersten Jahren des Internets lagen die Bandbreiten zwischen 9,6 und 56 kBit/s, deshalb wurde in diesen Zeiten sehr auf die Reduzierung der Datenmenge geachtet. Heutzutage hat sich die Bandbreite deutlich vergrößert und aufgrund dessen werden auch viele Ressourcen innerhalb einer Webapplikation genutzt (Kuhn/Raith 2013, 5). Abb. 1 veranschaulicht, wie mit der zunehmenden Größe aller Responses während des Transports die Anzahl von Requests steigt. Die Messungen wurden pro einer Webseite gemacht.

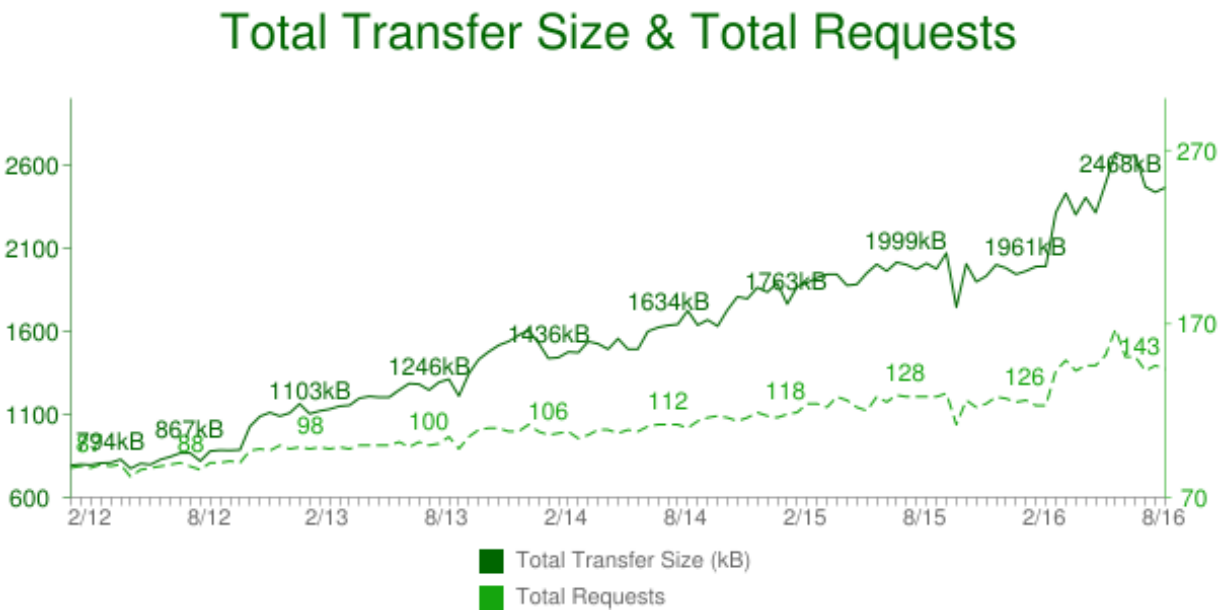

Abb. 1: Vergleich der durchschnittlichen Größe aller Responses während des Transports mit der durchschnittlichen Anzahl von Requests pro einer Webseite. Die Daten wurden zwischen Januar 2012 und August 2016 gemessen (<http://httparchive.org/>).

Mit der oben dargestellten Grafik kann man sagen, dass die Webapplikationen selbst immer komplexer werden und die Dateivolumen zunehmen werden. Deshalb kann gesagt werden, dass das Internet bis heute nicht schneller geworden ist, es können nur mehr Daten übertragen werden (Kuhn/Raith 2013, 5).

In diesem Zusammenhang ist die Webperformance – Optimierung sehr wichtig. Es werden ein paar Beispiele dafür genannt. Die Geschwindigkeit einer Webapplikation beeinflusst z.B. den Onlinehandel sehr. „Amazon" hat herausgefunden, dass jede 100 ms Verzögerung beim Laden der Webseite die Verkäufe um 1% verringert. „Google" und „Microsoft Bing" haben herausgefunden, dass 2 Sekunden Verzögerung beim Laden der Webseite das Einkommen um 4.3%

verringert. Die Untersuchungen der „Aberdeen Group“ zeigen, dass ein Viertel aller Nutzer nur drei Sekunden wartet, bis eine Webapplikation geladen hat. „Akamai“ hat ähnliche Untersuchungen gemacht und es hat sich herausgestellt, dass 57% aller Nutzer einer Webseite nach einer 4 Sekunden dauernden Wartezeit die Interaktion mit der Webseite beenden werden (Rigor, Inc. 2016). Außerdem kommt dazu, dass unabhängig von den Datenraten in mobilen Endgeräten, 71% der mobilen Nutzer erwarten, dass die Webapplikation nicht langsamer als am Desktop PC geladen wird (Equation Research 2011).

Etwa 80-90% der gesamten Geschwindigkeit der Webapplikation macht das Frontend aus. Diese Zahlen hat Steve Sounders, leitender Chefingenieur für Webperformance bei „Google“ herausgefunden. In seinen Forschungen konnte er zeigen, dass die große Mehrheit der Performance – Problemstellen, welche von den Websitenutzern gesehen werden, das Laden und Rendern von CSS, JavaScript Dateien und Bildern betrifft (Rigor, Inc. 2016). Deshalb sind die clientseitigen Techniken zur Performance – Optimierung sehr wichtig. In der aktuellen Masterarbeit wird der Fokus auf diese Techniken gelegt.

Man darf auch nicht vergessen, dass seit Juni 1999 für das Hypertext – Übertragungsprotokoll die Version HTTP/1.1 (Fielding et al. 1999) verwendet wurde. Wie zuvor gesagt wurde, sind die Webapplikationen mit deren vielfältigen Medieninhalten immer größer und komplexer geworden (Grigorik 2013, 161). Mit täglich zunehmenden Interaktionen im Web und der geforderten Ansprechbarkeit der Webapplikationen in Echtzeit, konnte die Leistung des zuvor verwendeten HTTP/1.1 – Protokolls kaum der benötigten Leistung entsprechen. Deshalb bestand ein Bedarf an zusätzlichen Modifikationen (Grigorik 2013, 162).

Im Januar 2012 wurde von der „HTTPbis Working Group“ eine Nachricht publiziert, dass sie eine neue Spezifikation des HTTP – Protokolls entwickeln werden. Die neue Version soll die Übertragungsleistung beschleunigen und gleichzeitig sollen weniger Latenz und eine größere Durchflussleistung entstehen. Gleichzeitig werden die Semantiken der höheren Ebene wie Methoden, Statuscodes, URIs und Headerfelder unverändert bleiben. Dies bedeutet, dass man die neue Version des Protokolls ohne zusätzliche Änderungen in der Webapplikation verwenden kann. (Grigorik 2013, 162)

Im Februar 2015 wurde die neue Version des HTTP – Protokolls: HTTP/2 von der IESG („The Internet Engineering Steering Group“) genehmigt (Grigorik 2015, 4). Im Mai des gleichen Jahres ist die offizielle Dokumentation des Protokolls erschienen: RFC – 7540 (Belshe et al. 2015) und RFC – 7541 (Peon/Ruellan 2015). Um den Ressourcenaustausch zwischen Client und Server unter dem bisher verwendeten HTTP/1 – Protokoll etwas zu beschleunigen, werden bestimmte Frontend – Optimierungsmaßnahmen für die Webapplikation angewendet. Diese werden oft als „Workarounds“ bezeichnet, weil diese viele Nachteile haben (Grigorik 2013, 162). Bei der Verwendung des neuen Protokolls sind die im HTTP/1-Protokoll beliebten „Workarounds“

nicht mehr nötig (Grigorik 2013, 242). Dies bedeutet, dass der Entwicklungsprozess einer Webapplikation unter dem HTTP/2 – Protokoll unkomplizierter sein sollte.

Mithilfe des HTTP/2 – Protokolls können existierende Problemstellen des HTTP/1 – Protokolls limitiert werden. Hinsichtlich der technologischen Möglichkeiten des HTTP/2 – Protokolls kann der Austausch zwischen dem Client und dem Server deutlich beschleunigt werden (Buch, Excerpt HTTP/2 IG, 5). Aus diesem Grund entstand die Hauptmotivation für die Untersuchung des neuen Protokolls, die in der aktuellen Masterarbeit schrittweise ausgearbeitet wird.

1.2. Zielsetzung

Das HTTP/2 – Protokoll verfügt über viele neue Funktionen, die die Ladezeit der Webapplikation rasant beschleunigen können (Buch, Excerpt HTTP/2 IG, 5). Einige Funktionen werden innerhalb der Benutzung dieses Protokolls automatisch angewendet. Bei anderen Funktionen besteht die Möglichkeit, diese für jede einzelne Webapplikation individuell anzupassen. Dazu gehört die Funktion „Server Push", die in zahlreichen Quellen (Grigorik 2015, 17), (Jayaprakash 2016), (Krasnov 2016), (Ross, 2016) als sehr leistungsfähig für die Beschleunigung der Ladezeit der Webapplikation erachtet wird. Wie zuvor erwähnt wurde, liegt der Fokus der aktuellen Masterarbeit auf den Untersuchungen der Frontend Performance – Optimierungstechniken. In diesem Zusammenhang ist es interessant, zu beobachten, wie groß die Einflüsse des „Server Pushs" auf die Ladezeit der Webapplikation sind. Welche Techniken müssen angewendet werden, um diese Funktion möglichst vorteilhaft nutzen zu können? Aufgrund dessen, liegt das erste Ziel dieser Masterarbeit auf der Untersuchung der Ladezeit der Webapplikation mithilfe des HTTP/2 – Protokolls und dessen Funktion „Server Push" in unterschiedlichen Einsätzen. Besonders wird dabei auf die clientseitigen Frontend Performance – Optimierungstechniken geachtet.

Das zweite Ziel ist der Vergleich der Unterschiede der Ladezeit der Webapplikation zwischen dem HTTP/1.1- und dem HTTP/2 – Protokoll, wenn die Webapplikation gesondert für jede Version des HTTP – Protokolls optimal angepasst wird. Es ist interessant zu untersuchen, wie groß die Vorteile des neuen Protokolls sind.

Das dritte Ziel, das in der aktuellen Arbeit verfolgt wird, ist die Auseinandersetzung mit der Konfiguration des Webservers für das HTTP/2 – Protokoll im Hinblick auf den „Server Push". Zum anderen sollen die wichtigsten Schritte und Metriken des kompletten Ladezeitprozesses einer Webapplikation ausgewählt werden, die sowohl für die Erstdarstellung, als auch für die Gesamtladezeit im Webbrowser wichtig sind.

Die Ergebnisse der aktuellen Arbeit sollen den Stand der aktuellen Implementierung des HTTP/2 – Protokolls zeigen. Dadurch sollen Frontend – Webentwicklern Hinweise gegeben werden, welche Maßnahmen unternommen werden müssen, um perfomante Webanwendun-

gen unter dem neuen Protokoll zu erstellen. Zum anderen werden angetroffene Schwierigkeiten und Schwachstellen aufgezeigt, die während des Testverlaufs oder der Testauswertung erkannt wurden.

1.3. Aufbau der Arbeit

Um die gesetzten Ziele zu erreichen, wurde die vorliegende Arbeit in mehrere Kapitel aufgeteilt: Theoretische Grundlagen, Vorbereitung der Tests und Testevaluation, Testdurchführung, sowie Evaluation und abschließende Betrachtung.

In ersten theoretischen Kapitel wird der kritische Rendering – Pfad und dessen Rolle in der Webperformance – Optimierung dargestellt. Es werden die Modelle der „Navigation Timing API" und der „Ressource Timing API" vorgestellt.

In einem weiteren theoretischen Kapitel wird zunächst analysiert, welche Problemstellen unter dem HTTP/1 – Protokoll existieren und mit welchen clientseitigen Techniken zur Performance – Optimierung diese gelöst werden können. Anschließend werden die wichtigsten Frontend – Optimierungstechniken beschrieben, die unabhängig von der Version des verwendeten HTTP – Protokolls für die Erstellung performanter Webapplikationen geeignet sind (sog. „Evergreen"-Techniken). Zuletzt wird als theoretische Grundlage das HTTP/2 – Protokoll und dessen Funktionen dargestellt. Ebenso werden die Frontend – Optimierungstechniken beschrieben, die für das neue Protokoll geeignet sein können.

Im Rahmen der Testvorbereitungen wird eine Webapplikation entwickelt, die für die Fragestellung dieser Masterarbeit interessant zum Untersuchen ist. Es werden jeweils für das HTTP/1.1- und HTTP/2 – Protokoll spezifische Frontend – Optimierungsmaßnahmen ausgewählt und auf die Webapplikation angewendet. Dadurch sollen optimale Bedingungen für die Webapplikation unter beiden Protokollen erreicht werden. Als nächstes wird erörtert, wie der passende Webserver ausgewählt und konfiguriert werden soll, damit er die Implementierung des HTTP/2 – Protokolls mit der Funktion „Server Push" unterstützt. Als letzten Schritt zur Testvorbereitung werden hilfreiche Tools für die Durchführung der Tests vorgestellt. Außerdem wird eine geeignete Auswertungsmethode vorgestellt.

Im nächsten Kapitel wird die entwickelte Webapplikation unter unterschiedlichen Bedingungen getestet und die Ergebnisse evaluiert. Da der Fokus dieser Arbeit auf der Untersuchung der Funktion „Server Push" liegt, werden dafür unterschiedliche Einsätze simuliert, um herauszufinden, welche dieser Einsätze für die Performanz vorteilhaft sein können.

Im zweiten Test-Teil wird untersucht, wie groß die Unterschiede der Ladezeit zwischen dem HTTP/1- und dem HTTP/2 – Protokoll sind. Während der Testevaluation werden die theoretischen Ergebnisse mit den erzielten Testergebnissen verglichen, um zu sehen, ob die theoretischen Annahmen zu den erzielten Ergebnissen passen.

Abschließend werden sämtliche Ergebnisse zusammengefasst und ein Ausblick gegeben.

2. Der kritische Rendering – Pfad und seine wichtigsten Komponenten

In der Einleitung wurde kurz erläutert, was die Web Performance für kommerzielle Webauftritte ausmacht und warum clientseitige Techniken zur Performance – Optimierung so wichtig sind. Performance – Optimierung ist ein Prozess, der zur Verbesserung der Ausliefergeschwindigkeit von Webdiensten führt, um so die Nutzererwartung (geringe Ladezeit) zu erfüllen (Mishunov 2015). Wie lange darf eine Webseite geladen werden? Nach dem „Platform Success Model", das von Google erstellt wurde (Duca/Glazkov 2016), ist das Limit der Ladezeit einer Webseite auf eine Sekunde begrenzt. Diese Reaktionszeit hängt mit psychologischen Aspekten der menschlichen Kommunikation zusammen. Normalerweise gibt es in einem persönlichen Dialog zwischen zwei Menschen eine kurze Reaktionszeit auf eine Frage. Nutzer wollen von einer Webseite die gleiche Reaktionszeit haben, wie bei zwischenmenschlicher Kommunikation. Aus diesem Grund muss man bei der Performance – Optimierung wichtige Zeitstufen aus der Psychologie im Kopf behalten (Mishunov 2015). Diese Stufen sind:

- „Augenblicklich": 0,1-0,2 Sekunden.
- „Unmittelbar": 0,5-1 Sekunde.
- „Phase der Absorption": 2 bis 5 Sekunden.
- „Endzeit der Aufmerksamkeit ": 5 bis 10 Sekunden.

(Mishunov 2015).

Wenn man diese Stufen auf das „Platform Success Model" anwendet, bekommt man Ladezeiten einer Webseite, die der Reaktionszeit zwischenmenschlicher Kommunikation ähnlich sind: der Ladevorgang der Webseite soll unmittelbar beginnen und die Nutzer sollen augenblickliche Rückmeldungen von der Webapplikation bekommen. Deshalb liegt das Ziel der Performance – Optimierung im Erreichen von Ladezeiten, die im Rahmen der Reaktionszeiten zwischenmenschlicher Kommunikation liegen (Mishunov 2015).

Allerdings zählen nicht nur die Zahlen, die beim Laden einer Webseite gemessen werden können. Wichtig ist auch die von Nutzern wahrgenommene Ladezeit. Diese besagt, was der Nutzer denkt, wie lange die Webseite geladen wurde. Um diese wahrgenommene Ladezeit optimieren zu können, müssen zuerst alle wichtigen (als erste sichtbare) Komponenten geladen werden. Wenn die Webseite z.B. einen Footer beinhaltet, deren Komponenten sich nicht im Viewport befinden, sind diese Informationen nur nebensächlich und können später geladen werden. Des Weiteren ist es auch wichtig, die Inhalte zu übergeben, mit denen der Nutzer interagieren kann. Man muss darauf achten, dass diese Inhalte auch eine Interaktionskomponente haben (z.B. klickbare Buttons). Der Schwerpunkt der Performance – Optimierung muss auf der wahrgenommenen Ladezeit liegen (Mishunov 2015).

In diesem Kapitel wird beschrieben, welche Schritte im Browser passieren, um eine Webapplikation darstellen zu können. Dazu wird erklärt, was der kritische Rendering – Pfad ist und warum es wichtig ist, diesen zu optimieren. Außerdem wird erläutert, wie eigentlich die Performance gemessen werden kann und welche Parameter dazu benötigt werden.

2.1. Aufbau des Document Object Model

Bevor der Browser erste Pixel der Webapplikation im Viewport darstellt, müssen einige Schritte stattgefunden haben. Das Document Object Model (DOM) der HTML Datei und das CSS Object Model (CSSOM) von allen CSS Dateien müssen aufgebaut und in einem Render – Baum zusammengefügt werden (Abb. 2) (Grigorik 2013, 168). Nachdem die ersten Bytes des HTML Dokuments den Browser erreichen, fängt er an, das HTML Dokument zu parsen (Grigorik 2016a).

Die DOM-Erstellung besteht aus mehreren Phasen: zuerst wird der Browser die Rohbytes mithilfe der Dateicodierung (z.B. UTF-8) in einzelne Zeichen übersetzen. Im nächsten Schritt werden die Zeichenfolgen, die sich in spitzen Klammern befinden, in eindeutige Token konvertiert, wie z.B. <html>, <head> u.s.w. Jedes Token hat gemäß dem „W3C" HTML5-Standard (W3C 2014a) eine spezielle Bedeutung und Satzregeln. Im dritten Schritt werden die Token in Objekte umgebaut, die eigene Merkmale und Eigenschaften haben. Im letzten Schritt werden die zuvor hergestellten Objekte für die Erstellung einer hierarchischen Baumstruktur verwendet (Grigorik 2016c).

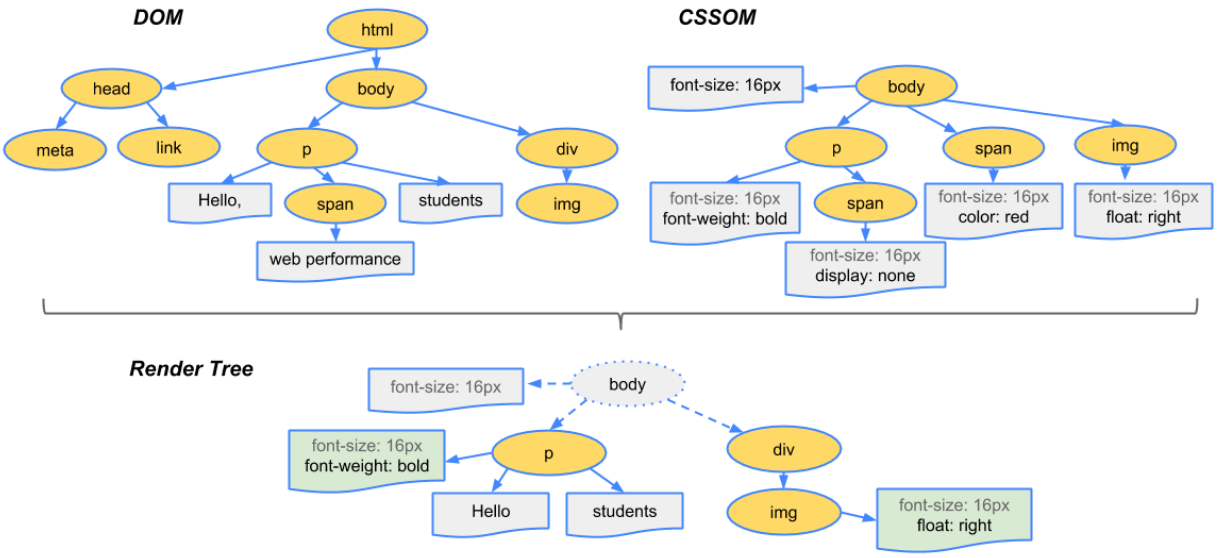

Abb. 2: Aufbau des Render - Baumes, (Grigorik 2016d).

2.2. Aufbau des CSS Object Model

Falls während dem HTML-Parsing ein Link-Tag auf eine externe CSS Datei gefunden wird, wird sie sofort am Server angefordert (Grigorik 2016c). Parallel zur DOM Erstellung wird das CSS Object Model (CSSOM) aufgebaut (Grigorik 2013, 168).

CSS Dateien sind auch für die Erstdarstellung der Webapplikation wichtig. Sie beschreiben die Formatierungsregeln, die für eine HTML Datei angewendet werden (Grigorik 2016d). Deshalb wird der Browser, wenn die CSS Datei vom Parser heruntergeladen wird, sofort anfangen, die CSS Datei zu bearbeiten. Die visuelle Darstellung wird dadurch blockiert werden (Grigorik 2016e).

Die CSSOM Erstellung besteht auch aus mehreren Phasen, die der DOM Erstellung sehr ähnlich sind (Abb. 2). Zuerst wird der Browser die Rohbytes in einzelne Zeichen mithilfe der Dateicodierung (z.B. UTF-8) übersetzen. Im nächsten Schritt werden die Zeichenfolgen in eindeutige Token konvertiert. Danach werden die Token in Objekte umgebaut, die eigene Merkmale und Eigenschaften von CSS Spezifikationen haben. Im letzten Schritt werden diese Objekte in einer Baumstruktur verbunden. CSSOM hat eine Baumstruktur, weil die CSS-Regeln zuerst auf ein Eltern-Element eines Objektes angewendet werden. Erst danach werden die CSS-Regeln auf das Objekt selbst angewendet (Grigorik 2016c).

2.3. Render – Baumstruktur

Auf der Basis der DOM- und CSSOM-Baumstrukturen wird eine Render – Baumstruktur erstellt (Abb. 2). Mithilfe der Render – Baumstruktur wird der Browser aus allen auf der Seite sichtbaren Elementen ein Layout erstellen. Nach diesem Schritt werden die ersten Pixel im Viewport erscheinen (Abb. 3) (Grigorik 2016c).

Der Prozess der Erstellung der Render – Baumstruktur besteht aus mehreren Etappen, die in Abb. 3 dargestellt werden:

1. Jeder sichtbare Knoten in der DOM – Baumstruktur wird gefunden. Dieser Schritt betrifft nicht alle Elemente, z.B. Script-Tags oder Metatags werden auf der Seite unsichtbar sein und werden weggelassen. Außerdem werden manche Elemente mithilfe der CSS-Regeln versteckt. Diese Elemente werden auch nicht in der Render – Baumstruktur erscheinen.
2. Für jeden Knoten, der sichtbar ist, werden entsprechende CSS-Regeln gesucht und angewendet.
3. Im letzten Schritt werden die sichtbaren Knoten mit den entsprechenden CSS-Regeln ausgegeben und die Render – Baumstruktur wird fertiggestellt.

(Grigorik 2016d).

Einfluss von JavaScript

JavaScript spielt auch eine große Rolle bei der Erstellung des DOMs und CSSOMs. JavaScript kann sowohl die Inhalte der HTML Seite als auch CSS-Regeln ändern und manipulieren (Abb. 3). Die DOM-Erstellung wird angehalten, bevor JavaScript nicht komplett heruntergeladen und ausgeführt wurde. Falls eine JavaScript Datei oder ein JavaScript-Code in der HTML Seite entdeckt wird, bevor der Aufbau des CSSOM fertig ist, wird der Browser warten, bis CSSOM komplett aufgebaut wird. Erst danach wird die JavaScript Datei ausgeführt. Dann kann das DOM fertig gestellt werden und der Browser kann mit dem Aufbau der Render – Baumstruktur anfangen (Grigorik 2016f), (Grigorik 2016h).

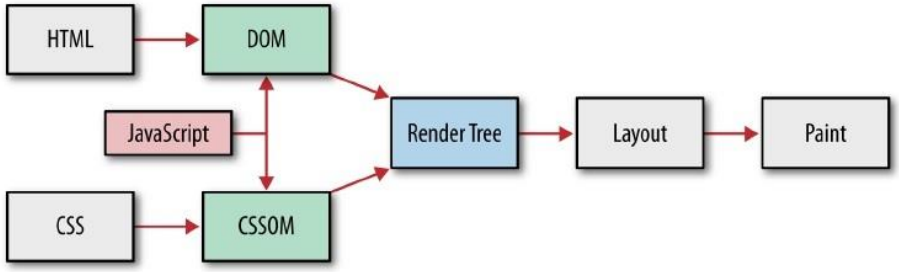

Abb. 3: Bearbeitungskette der HTML-, CSS- und JavaScript Dateien im Browser (Grigorik 2013, 168).

Für die Erstdarstellung auf dem Viewport sind nur HTML, CSS und JavaScript Dateien wichtig (Grigorik 2016h). Alle andere Dateien (z.B. Bilder oder Schriften) spielen hier keine Rolle. Die Ladereihenfolge von diesen Dateien ist wichtig: JavaScript muss so wenig wie möglich die Render – Baumstruktur verzögern. Falls die JavaScript Dateien synchron geladen werden, wird die DOM-Erstellung während des Ladens und der Ausführung der JavaScript Dateien angehalten (Grigorik, 2012). Gleichzeitig müssen CSS Dateien so schnell wie möglich dem Client übergeben werden, damit das CSSOM möglichst früh aufgebaut wird (Grigorik 2016h). Um dies zu erreichen, ist es empfehlenswert, CSS Dateien am Anfang der HTML Seite zu platzieren, damit der Browser gleich anfangen wird, diese herunterzuladen. Außerdem ist es hilfreich, Medienabfragen zu benutzen, in der bestimmte Merkmale zur CSS Datei stehen (z.B. media="print" für die Druckansicht). Mithilfe von diesen Merkmalen werden alle eingebundenen CSS Dateien heruntergeladen, aber die Ausführungszeit wird verkürzt (Grigorik 2016e).

JavaScript kann auf die DOM-Elemente zugreifen, deshalb müssen die DOM-Knoten zuerst erstellt werden. Außerdem müssen die DOM-Elemente im Baum gefunden werden, bevor JavaScript ausgeführt wird. Da JavaScript den weiteren DOM-Aufbau blockiert und gleichzeitig auf CSSOM wartet, ist es empfehlenswert, die JavaScript Dateien am Ende des HTML Dokuments einzubinden (Grigorik, 2012).

Wenn die JavaScript Datei für die DOM-Struktur und die Erstdarstellung nicht wichtig ist, ist es empfehlenswert, die Skripte asynchron zu laden oder dann zu laden, wenn das DOM fertig

gebaut ist. „Async" und „defer" Attribute sind boolean Attribute, die zeigen, wie die JavaScript Dateien ausgeführt werden sollen (W3C 2014b).

Wenn asynchrone JavaScript Dateien gefunden werden, werden sie sofort während dem HTML-Parsing heruntergeladen, und asynchron ausgeführt, sobald sie zur Verfügung stehen. Dies passiert unabhängig davon, in welcher Reihenfolge sie in der HTML Datei aufgelistet sind. Dafür ist ein „async" Attribut verantwortlich (W3C 2014b), (Souders 2010).

Eine andere Möglichkeit ist es, das Attribut „defer" zu verwenden. Dieses signalisiert dem Browser, dass er die JavaScript Datei erst ausführen muss, wenn das HTML Dokument fertig geparsed ist (W3C 2014b). JavaScript Dateien mit dem Attribut „defer" werden parallel heruntergeladen, aber werden genau in der Reihenfolge ausgeführt, in welcher sie im HTML-Dokument aufgelistet sind. Dies passiert unabhängig davon, in welcher Reihenfolge sie angekommen sind (Souders 2010).

Bei der Verwendung von sowohl „defer" als auch „async" Attributen, wird die DOM-Erstellung nicht mehr blockiert (Grigorik, 2012).

Diese beiden Attribute werden von populären modernen Browsern unterstützt (Mozilla Developer Network and individual contributors 2016a).

Die Anwendung sowohl von „defer" als auch von „async" Attributen für Skripte, die im HTML-Code eingefügt sind, hat keinen Effekt (Mozilla Developer Network and individual contributors 2016a).

Nachdem die Render – Baumstruktur fertiggestellt ist, wird das Layout berechnet (Abb. 3). Je nachdem, wie komplex die HTML und CSS Dateien sind, ob JavaScript DOM-Elemente anders platzieren wird oder CSSOM-Elemente ändern wird, wird die Berechnung des Seitenlayouts einige Zeit brauchen. Nachdem das Layout fertig gestellt wurde, wird der Browser auf den jeweiligen Ebenen Teile der visuellen Elemente zeichnen. Dazu gehören z.B. Farben, Schriften, Bilder u.s.w. Im letzten Schritt werden alle Ebenen mit dem darauf gezeichneten Content zusammengesetzt. Dieser Schritt ist wichtig, damit die Elemente, die im Viewport angezeigt werden, ihre Richtige Stelle haben werden und nicht überlappen werden (Lewis 2016).

Die Produktivität dieser letzten Schritte hängt oftmals von der Entwicklungsqualität der einzelnen Bestandteile ab: HTML, CSS und JavaScript. Es gibt viele Optimierungsmöglichkeiten, die die Ausführung dieser Schritte beschleunigen können. Allerdings werden Optimierungsmöglichkeiten ab der Erstellung des Render – Baumes im Rahmen dieser Masterarbeit nicht betrachtet, weil diese Art von Optimierungen zur Optimierung einzelner Dateien der Webanwendung gehören und nicht direkt mit der Optimierung der Dateiauslieferung zusammenhängen.

Alle diese Schritte, die zwischen dem Empfang der HTML, CSS und JavaScript Dateien passieren, wie schnell diese verarbeitet werden, und bis diese als gerenderte Pixel im Viewport erscheinen, ist der kritische Rendering – Pfad (Grigorik 2016g).

2.4. „Navigation Timing API“

Für die detaillierte Analyse des kritischen Rendering – Pfades einer Webapplikation im Webbrowser steht das „Navigation Timing API“ Tool zur Verfügung (Grigorik 2016a). Die „Navigation Timing API“ wurde vom „W3C“ (World Wide Web Consortium) (W3C 2016a) spezifiziert und dokumentiert.

„Navigation Timing API“ definiert ein Interface für Webapplikationen, das Messdaten zum kompletten Zeitraum, der für die Navigation über das komplette Dokument (HTML Datei) benötigt wird, sammelt und diese Daten zur Verfügung stellt (W3C 2016b). Die „Navigation Timing API“ wurde in allen populären mobilen und Desktopbrowsern implementiert (Mozilla Developer Network and individual contributors 2016b). Mithilfe dieser API kann der Browser die gesammelten Messdaten der verschiedenen Phasen des kritischen Rendering – Pfades zur Verfügung stellen (W3C 2016b), (Grigorik 2016a).

„Navigation Timing API“ wird vom „W3C“ als „Performance Navigation Timing“ Interface bezeichnet. Abb. 4 illustriert die Attribute des „Performance Navigation Timing“ Interfaces (W3C 2016b). Diese Attribute werden in Millisekunden ausgeliefert.

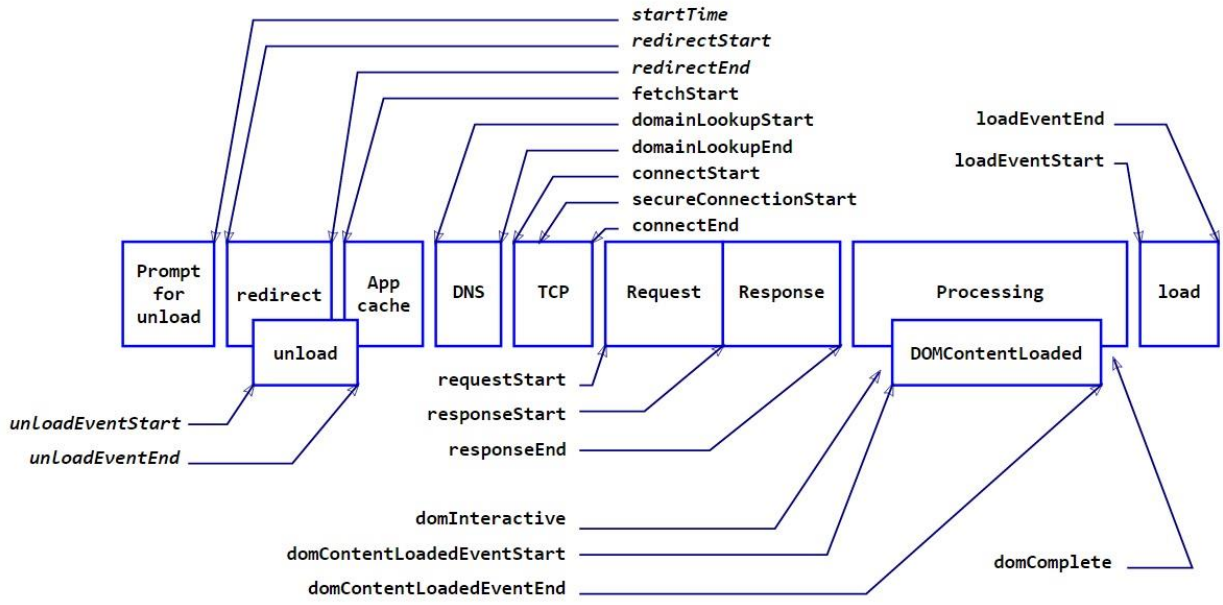

Abb. 4: „Performance Navigation Timing“ Interface (<https://www.w3.org/TR/navigation-timing-2/>).

Das „Performance Timing Interface“ stellt eine komplette Kette der Navigation dar, die in jedem Browser passiert. Jedes Performance Timing Attribut hat einen Start- und Endpunkt. Dies eröffnet die Möglichkeit, zu untersuchen, wie lange ein bestimmtes Event gedauert hat. Nach dem Aufruf einer Webapplikation wird zuerst die zuvor angezeigte Webapplikation vom Viewport gelöscht. Die Zeit dafür kann als Differenz zwischen „unloadEventEnd“ und „unloadEventStart“ berechnet werden. Falls es keine zuvor geladene Seite gibt, oder die frühere Seite oder einer der erforderlichen Redirects nicht die gleiche Domain haben, wird dieser Wert auf 0 gesetzt. Danach wird die Internetadresse vom Netzwerk aufgerufen (Mozilla Developer Network and individual contributors 2016c).

Attribute des „Performance Navigation Timing“ Interfaces	Bedeutung
„redirect“	Wird ausgegeben, wenn ein Redirect stattfindet.
„fetchStart“	Registriert den Zeitpunkt, an dem der Client anfängt, nach der Ressource mithilfe des HTTP-Requests zu fragen.
„domainLookupEnd“	Registriert den Zeitpunkt, an dem die IP-Adresse des Domainnamens bestimmt wird. Falls die IP-Adresse des Domainnamens aus dem Cache der Anwendung aufgerufen wird, wird dieser Parameter den gleichen Zeitpunkt wie der des „fetchStart“ Attributs haben.
„connectEnd“	Wird ausgegeben, wenn die Verbindung zum Server hergestellt wurde. Falls es eine verschlüsselte Verbindung ist, wird ein TSL-Handshake zwischen den „connectStart“- und „connectEnd“ Attributen angezeigt.
„requestStart“	Das HTML Dokument wird vom Server, dem Anwendungscache oder einer lokalen Ressource angefragt.
„responseStart“	Wird dann ausgegeben, wenn das erste Byte der Anfrage den Client erreicht.
„responseEnd“	Registriert den Zeitpunkt, an dem der Client alle Bytes der Anfrage bekommen hat.
„domLoading“	Zeigt den Zeitstempel, ab wann der Browser mit dem Parsen der empfangenen Bytes anfängt. Dieser Wert wurde in Abb. 4 nicht angezeigt, da alle existierenden Webbrowser auf unterschiedliche Weise das DOM erstellen und die Zeitpunkte für das „domLoading“ Attribut sich je nach Implementierung unterscheiden. Deshalb ist es nicht empfehlenswert, dieses Attribut für die Messungen zu benutzen (Grigorik 2016a), (W3C 2016b).

„domInteractive“	Das Attribut zeigt den Zeitstempel an, an dem das HTML-Dokument fertig geparsed wurde und das DOM erstellt wurde (Mozilla Developer Network and individual contributors 2016c) (Grigorik, 2012). In diesem Moment ändert sich der Document.readyState auf „interactive“. Dies bedeutet, dass das Dokument mit dem Laden fertig ist und dass es fertig geparsed wurde. Alle anderen Ressourcen, wie z.B. Styles, Bilder u.s.w. sind noch am Laden (Mozilla Developer Network and individual contributors 2016d).
„domContentLoadedEventStart“	Das Attribut markiert den Zeitpunkt kurz bevor der Parser das „domContentLoaded“-Event setzen wird. Dies passiert unmittelbar nachdem alle Skripte die für das Parsing ausgeführt werden müssen, ausgeführt wurden (Mozilla Developer Network and individual contributors 2016c). „DomContentLoaded“ ist ein Event das dann erscheinen wird, wenn das HTML Dokument komplett heruntergeladen und geparsed wurde, ohne auf Styles, Bilder u.s.w. zu warten (Mozilla Developer Network and individual contri-butors 2016e). Dieses Attribut wird in der jQuery JavaScript-Bibliothek für den Startpunkt der Ausführung von JavaScript Dateien benutzt, die innerhalb der „$(document).ready().“ geschrieben sind. In diesem Fall werden die Skripte ausgeführt, wenn das DOM komplett geladen wurde (The jQuery Foundation 2016a), (The jQuery Foundation 2016b).
„domContentLoadedEventEnd“	Das Attribut wird den Zeitpunkt markieren, an dem alle Skripte, die ausgeführt werden müssen, ausgeführt sind. Deshalb ist die Zeit, die zwischen domContentLoadedEventStart und domContentLoadedEventEnd benötigt wird – die Zeit für die Ausführung von JavaScript Dateien, die nach dem domContentLoadedEventStart Attribut ausgeführt werden müssen.
„domComplete“	Das Attribut wird gesetzt, wenn der Parser das Dokument fertig bearbeitet hat. In diesem Moment wird sich der Document.readyState auf den Status „complete“ ändern. Dies bedeutet, dass das Hauptdokument und alle anderen Ressourcen fertig geladen sind (Mozilla Developer Network and individual contributors 2016d).
„loadingEventStart“ und „loadingEventEnd“	Registriert den Zeitpunkt, an dem das „load“-Event für das Dokument gesetzt wurde. Das „load“-Event wird erscheinen, wenn das Dokument und alle davon abhängigen Ressourcen fertig

	geladen wurden (Mozilla Developer Network and individual contributors 2016f). Das „loadingEventEnd" Attribut wird den Zeitstempel anzeigen, an dem das „load"-Event abgeschlossen wird (Mozilla Developer Network and individual contributors 2016c). Dies ist der letzte Schritt beim Laden der Webapplikation: der Webbrowser wird das „onLoad" Ereignis ausgeben. Dies bedeutet, dass die weitere Anwendungslogik gestartet werden kann (Grigorik 2016a).

Tab. 1: Attribute des „Performance Navigation Timing" Interfaces (Mozilla Developer Network and individual contributors 2016c).

JavaScript Dateien, die das „defer" Attribut haben, werden gleich nach dem „Interactive"-Status ausgeführt und folglich das domContentLoaded Event verzögern. Weil die Skripte gleich nach diesem Status ausgeführt werden, besteht noch die Gefahr, dass das CSSOM nicht fertig gebaut ist. Dies bedeutet, dass alle JavaScript Dateien, die ausgeführt werden müssen, auf die Fertigstellung des CSSOMs warten müssen. JavaScript Dateien, die das „async" Attribut haben, werden nicht das domContentLoaded-Event verzögern (Grigorik, 2012).

2.5. „Resource Timing API"

Die „Navigation Timing API" liefert die Messdaten für das ganze Dokument. Allerdings ist es oftmals für die komplette Analyse nicht ausreichend und es ist notwendig, um Messdaten für einzelne Ressourcen zu bekommen. Dafür wurde die „Resource Timing API" entwickelt. Die „Resource Timing API" ist eine JavaScript API, die detaillierte Informationen über Webapplikationsressourcen, während diese geladen werden, in Form von Zeitstempeln aus dem Netzwerk ausliest (Abb. 5) (Mozilla Developer Network and individual contributors 2016h).

Die Attribute der „Resource Timing API" liefern detaillierte Informationen zu Netzwerkevents wie Start- und Endzeiten von „redirect", „fetchStart", Start- und Endzeiten von DNS-Lookup oder Start- und Endzeiten von Request und Response. Außerdem wurden zusätzliche Parameter unterstützt, die Informationen der einzelnen Dateien über den Typ, die Dateigröße, die übertragene Dateigröße u.s.w. zur Verfügung stellen (Mozilla Developer Network and individual contributors 2016h), (W3C 2016c).

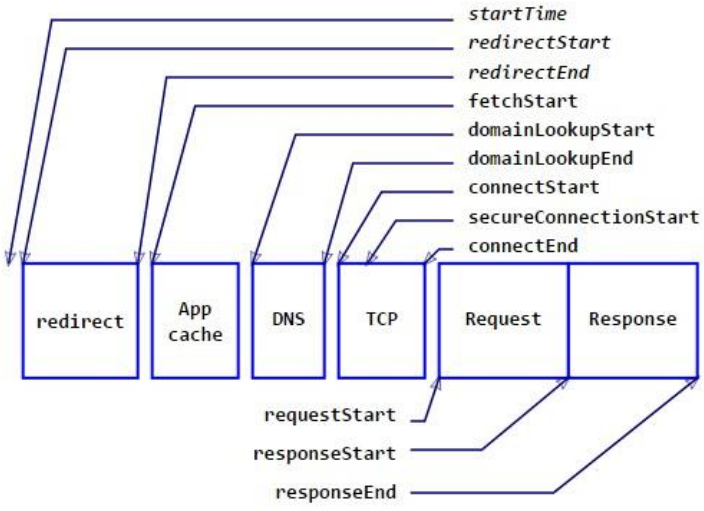

Abb. 5: „Resource Timing API" (<https://www.w3.org/TR/resource-timing/>).

Viele Entwicklertools von Browsern unterstützen sowohl die „Navigation Timing API", als auch die „Resource Timing API" (Mozilla Developer Network and individual contributors 2016g). Da die Attribute, die die „Resource Timing API" beinhaltet, ähnlich zu den Attributen der „Navigation Timing API" sind (mit einem einzelnen Unterschied, nämlich dass die Zeitstempel im Fall der „Resource Timing API" zu einzelnen Ressourcen gehören), werden sie nicht noch Mal beschrieben. Im Network Panel, das sich im Browser-Entwicklungstool befindet, befinden sich mehrere Parameter, die die „Resource Timing API" interpretiert. Das Zeitintervall von diesen Parametern liegt meistens im Millisekundenbereich. Die Anzahl dieser Parameter unterscheiden sich ein wenig je nach Webbrowser. Für alle gilt:

- DNS-Auflösung.
  Stellt die benötigte Zeit für die DNS-Auflösung dar (Garbee 2016).
- Verbindungsaufbau
  Zeigt, wie lange der Verbindungsaufbau von TCP- und SSL-Roundtrips gebraucht hat (Garbee 2016).
- Request Senden.
  Zeigt, wie lange der Request zum Server gedauert hat. (Garbee 2016).
- Warten.
  Warten ist auch als „Time To First Byte" bekannt. Diese beinhaltet die Zeit für den Roundtrip zum Server und die serverseitige Wartezeit für die Requestsbearbeitung (Garbee 2016).
- Herunterladen.
  Beinhaltet die Zeit für den Empfang des Responses (Garbee 2016).

(Microsoft 2016), (Apple 2016), (Mozilla Developer Network and individual contributors 2016i), (OPERA SOFTWARE ASA 2016), (Garbee 2016).

2.6. Zwischenfazit: Welche Parameter sind für die Untersuchung des kritischen Rendering – Pfades wichtig?

In dieser Masterarbeit wird der Fokus der Untersuchung auf der Dateiauslieferung und deren Art liegen. Dies bedeutet, dass in der Kette des kritischen Rendering – Pfades vor allem der Teil untersucht wird, der vor der Erstellung des Render – Baums liegt (Abb. 3).

Um die Dateiauslieferung genauer zu prüfen, sind viele Faktoren wichtig: wie schnell baut sich die Verbindung zum Server auf, wie schnell werden die wichtigsten Dateien für die Erstdarstellung zum Client geliefert und wie schnell werden sich CSSOM und DOM aufbauen. Diese Faktoren beeinflussen die Zeit, die vergeht, bis die ersten Pixel im Viewport erscheinen werden und wie lange die Webapplikation benötigt, um alle dazugehörigen Ressourcen vom Server anzufragen und herunterzuladen.

Um diese Faktoren prüfen zu können, werden mehrere Parameter der aufgerufenen Webapplikation geprüft. Dafür werden sowohl die Parameter aus der „Navigation Timing API" für die komplette Seite der Webapplikation als auch die Parameter aus der „Resource Timing API" für einzelne Ressourcen verglichen.

Interessante Parameter für diese Untersuchung sind:

- Zeit bis zum Aufbau der Verbindung zum Server (DNS-Lookup, TCP-Verbindung, SSL-Verbindung).
- Wie viele Bytes der Browser für jede Anfrage zum Server benötigt hat.
- „RequestStart", um zu schauen, wann die Anfrage gestartet wurde.
- „Time To First Byte", um zu schauen, wieviel Zeit zwischen dem Aufruf der Datei bis zur Ausgabe der ersten Bytes dieser Datei an den Client vergeht.
- „domInteractive", um zu schauen, wieviel Zeit zwischen dem Aufruf der HTML Datei und der Fertigstellung des DOMs vergangen ist.
- Zeitpunkt, an dem der Webbrowser anfängt, etwas im Viewport zu zeichnen.
- „domComplete", um zu schauen, wie lange es gebraucht hat, bis alle zur Webapplikation gehörigen Ressourcen heruntergeladen sind.

In der „Navigation Timing API" gibt es leider keine Möglichkeit, um zu beobachten, wann das CSSOM fertig ist. Man kann nur schauen, ab welchem Zeitpunkt im Viewport etwas gezeichnet wird und davon ausgehen, dass kurz zuvor das CSSOM fertig gestellt wurde.

3. Die Probleme des HTTP/1.1 – Protokolls, das HTTP/2 – Protokoll und deren Optimierungsmöglichkeiten.

3.1. HTTP/1.1 – Optimierungstechniken, Stand bisher

Unter dem HTTP/1.1 – Protokoll wurden mehrere Frontend Performance – Optimierungstechniken verwendet, die die Einschränkungen dieses Protokolls versuchen aufzuheben (Grigorik 2013, 188). Diese sind meistens sogenannte „Workarounds", weil manche von diesen Maßnahmen für die Entwicklung einer Webapplikation unnötig sind (Grigorik 2013, 188) und teilweise nicht erwünschte Nebeneffekte mit sich bringen. Die meist verbreiteten davon sind:

- Nutzung von mehreren TCP – Verbindungen.
- Aufteilung von Ressourcen auf mehrere Domains („Domain Sharding").
- Zusammenfassung von Ressourcen und „Spriting" von Bildern.
- Ergänzung von Code in der HTML Datei („Ressource Inlining").

(Grigorik 2013, 188).

Im Vergleich zum HTTP/1.0 – Protokoll sind im HTTP/1.1 – Protokoll mehrere nützliche Funktionen erschienen, die für die Performance einer Webapplikation sehr wichtig sind. Eine davon ist die Funktion „Keepalive Connections", um mehrere Requests innerhalb einer TCP – Verbindung bearbeiten zu können (Grigorik 2013, 159-160). Der Aufbau jeder TCP – Verbindung beginnt wie immer mit dem „Drei-Wege-Handschlag", aber dieser muss dank der Funktion „Keepalive Connections" nur einmal aufgebaut werden (Abb. 6) (Grigorik 2013, 189).

Das HTTP/1.1 – Protokoll ist so aufgebaut, dass innerhalb einer TCP – Verbindung nur ein Request oder Response innerhalb eines Zeitraums bearbeitet werden kann: zuerst wird der Request an den Server verschickt, dann wird dieser bearbeitet und der Response wird zurückgeschickt. Danach wird der nächste Request von der clientseitigen Warteschlange abgeschickt. Aus diesem Grund ist die Bearbeitung von Requests und Responses nicht wirklich parallel. Allerdings wurde im HTTP/1.1 – Protokoll die Funktion „HTTP Pipelining" hinzugefügt, die die eben genannte Warteschlange vom Client zum Server verschiebt. Damit werden nicht die Requests am Client warten, sondern die Responses werden am Server in die Warteschlange gestellt. Wenn man die übliche Bearbeitungsweise ohne die Funktion „HTTP Pipelining" betrachtet, sieht man, dass auf der Serverseite zwischen dem Versand eines Responses vom Server und der Ankunft des nächsten Requests vom Client nichts gemacht wird (Abb. 6).

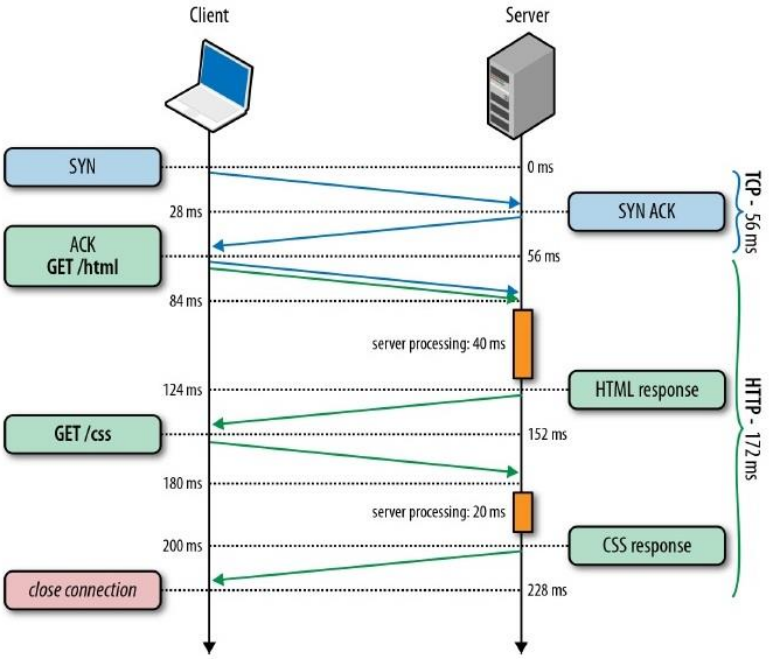

Abb. 6: Übliche Verarbeitungsweise der Requests und Responses innerhalb einer TCP – Verbindung mit der Funktion „Keepalive Connections" (Grigorik 2013, 191).

Mithilfe der Funktion „HTTP Pipelining" wird diese Zeit effizienter genutzt, wenn mehrere Anfragen an den Server gleichzeitig geschickt werden und der Server entsprechende Responses nacheinander liefern wird (Abb. 7). Dadurch wird zusätzlicher „Roundtrip" zwischen den Requests gespart (Grigorik 2013, 193).

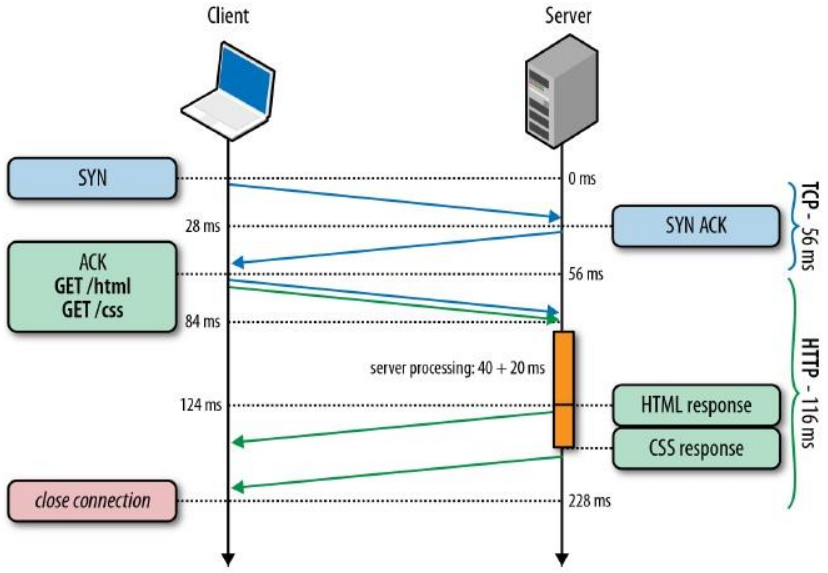

Abb. 7: Verarbeitungsweise der Requests und Responses innerhalb einer TCP – Verbindung mit den Funktionen „Keepalive Connections" und „HTTP Pipelining" (Grigorik 2013, 193).

Die Funktion „HTTP Pipelining" hebt die Einschränkungen des HTTP/1.1 – Protokolls leider nicht völlig auf: die Responses werden innerhalb einer TCP – Verbindung nacheinander geliefert. Bevor nicht alle Bytes des zuvor gesendeten Responses den Client erreichen werden, wird kein neuer Response vom Server gesendet. Es gibt keine Möglichkeit, um die Responses während der Lieferung miteinander zu multiplexieren (Grigorik 2013, 194). Außerdem hat diese Funktion nicht nur Vorteile: wenn HTML und CSS Dateien gleichzeitig am Server bearbeitet werden und die CSS Datei schneller zur Verfügung stehen wird, muss diese trotzdem warten, bis die HTML Datei am Client zugestellt wird. Dies ist als „Head-Of-Line Blocking" – Problem bekannt und verursacht ein schlechtes Szenario: die Lieferung der Dateien wird deutlich verzögert, wenn die Bearbeitung früher gesendeter Requests zu lange braucht und der Server verbraucht seinen Puffer, wodurch er überlastet werden kann. Außerdem werden TCP – Pakete eines Responses, wenn sie verloren gehen, erneut versendet. Dadurch kann die Verarbeitungszeit der Gesamtlieferung verdoppelt werden (Grigorik 2013, 195).

Noch ein Problem der Funktion „HTTP Pipelining" besteht darin, dass die meisten Browser diese nicht unterstützen (Grigorik 2013, 195), (Wikipedia 2015).

Aus oben genannten Gründen wird die Funktion „HTTP Pipelining" als Performance – Optimierungstechnik für das HTTP/1.1 – Protokoll nicht mitgezählt (Grigorik 2013, 195).

3.1.1. Nutzung von mehreren TCP – Verbindungen

Innerhalb des HTTP/1.1 – Protokolls wurde die Funktion „Keepalive Connections" hinzugefügt. Dies macht die Webapplikationen viel perfomanter, als noch unter dem HTTP/1.0 – Protokoll (Grigorik 2013, 187). Allerdings ist die Verwendung von nur einer TCP – Verbindung oftmals nicht genügend, weil dies zu langsam ist. Aus diesem Grund wurde sowohl client- als auch serverseitig erlaubt, mehrere TCP – Verbindungen per Hostname innerhalb einer Sitzung parallel zu unterstützen (Grigorik 2013, 196). Die genaue Anzahl unterscheidet sich je nach Browser- und Serverimplementierung. In der Standardkonfiguration sind bis zu sechs parallele TCP – Verbindungen erlaubt (Grigorik 2013, 196). Durch mehrere gleichzeitig geöffnete TCP – Verbindungen wird die Ladezeit der Webapplikation deutlich verringert. Dabei spielt auch die Funktion des TCP – Protokolls „Slow Start" eine nicht so große Rolle (siehe unten die genauere Beschreibung des TCP – Protokolls) (Grigorik 2013, 196).

Andererseits zieht die Parallelisierung von Responses einige Schwierigkeiten mit sich. Obwohl die Ressourcen parallel heruntergeladen werden können, teilen alle geöffneten TCP – Verbindungen eine Bandbreite. Außerdem werden mehr Puffer- als auch CPU- Kapazitäten auf beiden Seiten der Kommunikation verbraucht, je mehr Verbindungen geöffnet werden. Die Nutzung von mehreren TCP – Verbindungen ist eine Art, um die Requests und Responses zu parallelisieren, aber diese bleibt letztlich nur ein „Workaround" auf Grund der Einschränkungen des HTTP/1.1 – Protokolls (Grigorik 2013, 197).

Funktionsweise des TCP – Protokolls

Innerhalb des TCP – Protokolls werden Dateneinheiten zwischen Sender und Empfänger ausgetauscht, die als „TCP – Segmente" bezeichnet werden. Jedes „TCP – Segment" beinhaltet einen mind. 20 Byte großen Protokollkopf („TCP – Header") und die zu übertragenden Daten. Der „TCP – Header" besteht aus mehreren Feldern (Holtkamp 2001).

Eines der wichtigsten davon ist das „Window – Feld", das für die Flusssteuerung („Flow Control") verantwortlich ist. Dieses Feld ist wichtig, weil dieses die Anzahl von Bytes beinhaltet, die beim Empfänger ab dem ersten bestätigten Byte aufgenommen werden können. Mit dieser Mitteilung der Fenstergröße kann das TCP – Protokoll die Flusssteuerung („Flow Control") fortsetzen. Das TCP – Protokoll funktioniert nach der Methode „Sliding Window" mit variabler Größe. Damit kann jede Seite der Kommunikation so viele Bytes senden, wie bei der Fenstergröße angegeben sind. Dadurch entsteht beim Empfänger keine Warteschlange an Daten (Holtkamp 2001).

Innerhalb jeder TCP – Verbindung verfügt das TCP – Protokoll über den „cwnd" Algorithmus („Congestion window size"), der die Anzahl von TCP – Segmenten kontrolliert, die zwischen dem Sender und dem Empfänger ausgetauscht werden können. „Cwnd" wird serverseitig die Daten zum Senden begrenzen, bevor ihn nicht die Antwort vom Empfänger erreichen wird, dass die Daten erfolgreich zugestellt wurden (Grigorik 2013, 19). „Cwnd" funktioniert ähnlich wie der Mechanismus der Flusssteuerung („Flow Control") und ist dafür da, um den Traffic unter Kontrolle zu halten (Wikipedia 2016d).

Weder Client noch Server kennen die Bandbreite der gerade geöffneten TCP – Verbindung. Aus diesem Grund wird ein Mechanismus gebraucht, um die Flusssteuerung während einer bestehenden TCP –Verbindung zu kontrollieren (Grigorik 2013, 19). Mithilfe der TCP – Funktion „Slow – Start" kann während der bestehenden TCP – Verbindung der „cwnd" Algorithmus gesteuert werden. Sobald die Verbindung aufgebaut wird, wird „cwnd" auf „Maximum Segment Size" gesetzt. Wenn mit dieser gesetzten Größe alle TCP – Pakete den Empfänger erfolgreich erreichen werden, wird „cwnd" für die nächste „Round-Trip Time" verdoppelt werden. Wenn die TCP – Pakete innerhalb der Verbindung dagegen verloren gehen, wird „cwnd" verringert (Abb. 8) (Wikipedia 2016d), (Grigorik 2013, 20). Je kleiner also die Transaktionen zwischen Client und Server sind, desto sicherer wird es, dass die Daten schneller vollständig den Client erreichen werden (Grigorik 2013, 34). Aus diesem Grund kann man sagen, dass, je kleiner Ressourcen gehalten werden, desto schneller werden diese am Client ankommen.

Für TCP „Slow Start" wurde im April 2013 die „cwnd" – Startgröße von 10 „TCP – Segmenten" vom „RFC 6928" festgelegt (Grigorik 2013, 20).

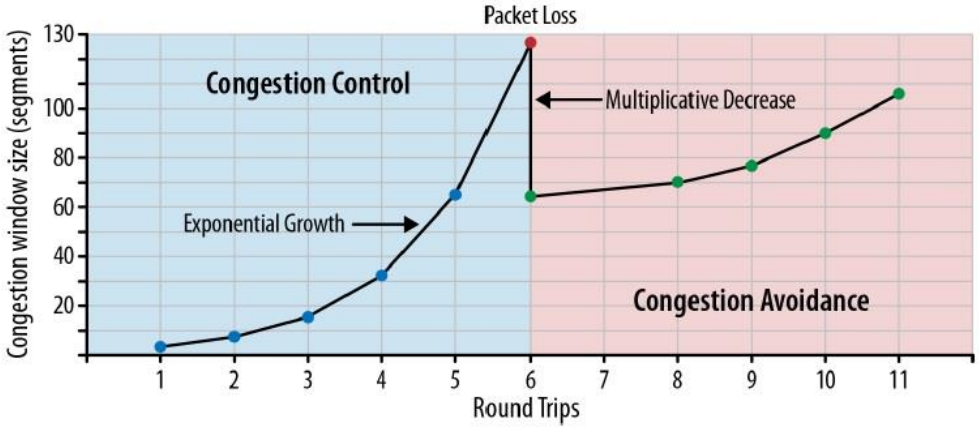

Abb. 8: Datenaustauschkontrolle der TCP – Verbindung, Funktionsweise des „Slow – Start" (Grigorik 2013, 20).

Aus der oben beschriebenen Funktionsweise des TCP – Protokolls kann man sagen, dass die Funktion „Slow – Start" eine wichtige Rolle bei der Performance der Webapplikation spielt. Das HTTP – Protokoll läuft über das TCP – Protokoll und unabhängig von der Größe der Bandbreite, startet jede TCP – Verbindung mit „Slow – Start". Aus diesem Grund ist es wichtig, um am Anfang die „cwnd" – Startgröße auf 10 „TCP – Segmenten" zu halten (Grigorik 2013, 20, 22).

3.1.2. Aufteilung von Ressourcen auf mehrere Domains („Domain Sharding")

Die Anzahl von parallel laufenden TCP – Verbindungen per Hostname innerhalb des HTTP/1.1 – Protokolls wird vom Client bestimmt (unter der Bedingung, dass die Serverimplementierung diese auch unterstützt). Normalerweise werden bis zu sechs parallele TCP – Verbindungen geöffnet. In manchen Situationen (wenn die Webapplikation über viele Ressourcen verfügt) kann aber diese begrenzte Anzahl von Verbindungen zu wenig sein (Grigorik 2013, 198-199).

Aus diesem Grund entstand noch ein „Workaround": Aufteilung von Ressourcen auf mehrere Domains, um eine noch größere Parallelisierung zu bekommen. Die Ressourcen können auf unterschiedliche Subdomains aufgeteilt werden, wie z.B.: {shard1, shard2}.example.com (Grigorik 2013, 198). Mithilfe dieser Technik erscheint es möglich, dass die Ressourcen schneller geladen werden.

Allerdings ist zu beachten, dass der Verbindungsaufbau zu jeder neuen Domain gleichfalls den Aufbau von DNS – Lookup erfordert. Dies kann je nach Situation viel Zeit kosten. Diese Optimierungstechnik im Zusammenhang mit der Funktion des TCP – Protokolls „Slow Start" kann negative Auswirkungen bei Nutzung von mobilen Netzwerken (3G und 4G) haben. Außerdem muss während der Entwicklung der Webapplikation genau beachtet werden, wie die Ressourcen aufgeteilt werden (Grigorik 2013, 199-200).

3.1.3. Zusammenfassung von Ressourcen und „Spriting“ von Bildern

Der schnellste Request ist derjenige Request, der nicht gemacht wird. Die Reduzierung der Anzahl der gemachten Requests ist die beste Performance – Optimierungstechnik. Diese Regel gilt für alle HTTP – Protokolle, unabhängig von deren Version. Besonders wenn die Ressourcen innerhalb einer TCP – Verbindung unter dem HTTP/1.1 – Protokoll nacheinander geliefert werden, und sechs TCP – Verbindungen geöffnet werden können, ist die Reduzierung von Anfragen eine gute Optimierungstechnik. Aus diesem Grund wird noch eine Performance – Optimierungstechnik für das HTTP/1.1 Protokoll angewendet: mehrere Ressourcen eines Typs werden zusammengefasst (Grigorik 2013, 196, 201).

Diese Technik betrifft vor allem CSS und JavaScript Dateien, sowie Bilder. Sowohl JavaScript Code als auch CSS – Regeln kann man problemlos zusammenfassen. Auch mehrere Bilder lassen sich als ein größeres Bild zusammenbinden.

Es gibt ein paar Vorteile dieser Technik:

- Bestehende Netzwerklatenz wird reduziert, weil weniger Requests und Responses zwischen Client und Server ausgetauscht werden.
- Je weniger nicht komprimierte HTTP – Kopfzeilen (Headers) ausgetauscht werden, desto mehr ausgetauschte Bytes werden gespart. HTTP – Kopfzeilen werden während dem Transport unter dem HTTP/1.1 – Protokoll nicht komprimiert.

(Grigorik 2013, 202).

Allerdings, wie alle bestehenden „Workarounds“ für die Performance – Optimierung für das HTTP/1.1 – Protokoll Nachteile haben, hat auch diese oben beschriebene Technik Nachteile. Zuerst müssen die Entwickler eine zusätzliche Maßnahme unternehmen, um entwickelte Applikationsmodule zusammenzufassen. Im Fall von „Sprites“ (Zusammenfassung von mehreren Bildern) muss man oft zusätzliche CSS – Regeln schreiben. Diese Technik wird dann gebraucht, wenn z.B. beim „Hover“ auf ein bestimmtes Bild ein Ereignis passieren soll. Da alle Bilder in ein großes Bild zusammengefasst sind, muss man diese künstlich wieder aufteilen. Dies braucht zusätzliche Zeit für die Entwicklung der Webapplikation (Grigorik 2013, 202).

Es gibt noch weitere wichtige Nachteile bei der Zusammenfassung von Ressourcen. Alle gebundenen Ressourcen werden eine gemeinsame URL haben. Dadurch wird es unmöglich, um einzelne Dateien zu cachen. Deshalb wird die Ressource nicht mehr vom Cache aufgerufen und muss neu geladen werden, wenn nur eine Kleinigkeit innerhalb der Datei der zusammengefassten Ressourcen durchgeführt wird (Grigorik 2013, 202).

Außerdem müssen alle dazugehörigen Ressourcen geladen werden, wenn nur eine der zusammengefassten Ressourcen auf einer konkreten Internetseite gebraucht wird.

Hinzu kommt auch, dass größere Dateien unter dem HTTP/1.1 – Protokoll deutlich später geladen werden. Dadurch werden alle anderen Ressourcen blockiert (Grigorik 2013, 202).

Der letzte wichtige Nachteil dieser Performance – Optimierungstechnik betrifft die zusammengefassten Bilder („Sprites"). Wenn von dem großen Bild, das durch Zusammenfassung entstanden ist, nur ein kleiner Bereich auf der Seite angezeigt wird, muss das komplette Bild trotzdem im RAM des Browsers gehalten werden. Besonders stark sind mobile Endgeräte davon betroffen (Grigorik 2013, 203).

3.1.4. Ergänzung des Codes in der HTML Datei („Ressource Inlining")

Requests unter dem HTTP/1.1 – Protokoll können viel Zeit in Anspruch nehmen (Grigorik 2013, 202). Aus diesem Grund können Ressourcen direkt in der HTML Datei hinzugefügt werden. Dies betrifft fast alle Ressourcentypen: CSS, JavaScript, Bilder, Audio und PDF. Die drei letzten Ressourcentypen können durch „Data – URI" hinzugefügt werden (Grigorik 2013, 204).

„Data – URI" ist ein „URI – Schema", mithilfe dessen die Daten dem Quelltext hinzugefügt werden können (Wikipedia 2016e). „Data – URI" benutzt das „base64" – Kodierungsschema, das für die Einbettung von binären Dateien gedacht war (McLachlan 2013). Deshalb ist es möglich, um binäre Dateien direkt in den HTML, CSS und JavaScript Dateien einzubetten. Das Bild wird dazu in einen „Base64 – String" konvertiert (Kuhn/Raith 2013, 205).

Der Vorteil von „Data – URI" besteht darin, dass kein zusätzlicher Request benötigt wird. Nach McLachlan 2013 zählen zu den Nachteilen:

- „Data – URIs" können nicht separat von dem Dokument (HTML, JavaScript oder CSS) gecacht werden, sie werden immer neu geladen.
- Der Browser benötigt mehr Zeit, um den „base64" – Code zu dekodieren.
- In den mobilen Endgeräten ist die Bearbeitung von „Data – URIs" bis zu 6 Mal langsamer.

Außerdem sind „base64" – kodierte Daten 1/3 größer als ihre binären Äquivalente (Grigorik 2013, 205).

Aus oben genannten Gründen kann man sagen, dass „Data – URIs" nicht immer vorteilhaft sind. Data – URI" kann aber für die Ergänzung kleinerer binären Ressourcen nützlich werden (Grigorik 2013, 204).

Alle Textbasierten Ressourcen wie CSS und JavaScript können in der HTML – Datei hinzugefügt werden. Dadurch wird die Netzwerklatenz durch Reduzierung von Requests verbessert (Grigorik 2013, 205). Dies kann auch dann praktisch sein, wenn Ressourcen für die Erstdarstellung wichtig sind.

Aber es existieren auch für diese Webperformance – Technik mehrere Nachteile. Wenn die Ressourcen hinzugefügt werden, wird keine Möglichkeit mehr bestehen, um diese individuell dem

Browsercache zu übergeben. Auch besteht die Gefahr, dass hinzugefügte Ressourcen auf unterschiedlichen Seiten benutzt werden, auch wenn diese dort nicht nötig sind. Die Größe der HTML Datei wird dadurch auch zunehmen (Grigorik 2013, 205).

3.2. „Evergreen" Frontend – Optimierungsmöglichkeiten

Unabhängig davon, welche Version des Netzwerkprotokolls verwendet wird, müssen alle Webapplikationen nach einer Reduzierung unnötiger Netzwerklatenz und der Verringerung von transportierten Bytes streben. Diese beiden Kriterien gehören zu den „Best Practices" der Webperformance und sind Basis für viele weitere Optimierungen. Die wichtigsten Regeln dabei sind:

- Reduzierung von DNS – Lookups und Domainweiterleitungen.
  Jede Auflösung des Hostnamens braucht einen „Roundtrip" aus dem Netzwerk, bis die IP – Adresse des dazugehörigen Hostnamens ermittelt wird (Grigorik 2013, 199-200). Dies passiert, bevor der erste Request zum Server geschickt werden kann. Je mehr Domains für das vollständige Laden der Webapplikation benötigt werden, desto größer ist die Verzögerung zum Herunterladen der Ressourcen. Dies betrifft auch die Domainweiterleitungen (Grigorik 2013, 235).
- Wiederverwendung aufgebauter TCP – Verbindungen.
  Da der Aufbau jeder TCP – Verbindung mit dem „Drei-Wege-Handschlag" beginnt und das TCP – Protokoll am Anfang nur eine geringere Anzahl von TCP – Segmenten erlaubt, sollten aufgebaute TCP – Verbindungen wieder verwendet werden (Grigorik 2013, 19), (Grigorik 2013, 235).
- Verwendung des CDNs (Content Delivery Network).
  Je näher sich die Clients geographisch zum Server befinden, von dem die Ressourcen heruntergeladen werden, desto schneller wird die Webapplikation geladen. Der wesentliche Grund dafür ist, dass die Verbindungsaufbauzeit zum Server deutlich verringert werden kann. CDNs können sowohl für statischen als auch für dynamischen Content benutzt werden (Grigorik 2013, 235).
- Reduzierung von unnötigen Ressourcen.
  Je weniger Requests benutzt werden, desto schneller wird die Webapplikation geladen (Grigorik 2013, 235).
- Nutzung des Browser-Cachings.
  Damit die Ressourcen nicht bei jedem Aufruf der Webapplikation immer wieder neu geladen werden, müssen diese die Fähigkeit haben, um einige Zeit im Cache des Browsers zu bleiben (Grigorik 2013, 236).
- Datenkompression während des Transports.
  Ressourcen der Webapplikation müssen mit minimaler Anzahl von Bytes transportiert werden. Aus diesem Grund muss eine Kompressionsmethode während des Datentransports angewendet werden (z.B. „Gzip") (Grigorik 2013, 236-237).

— Reduzierung der Datenmenge der Ressourcen.
  Die Datenmenge für die Webapplikation kann durch Reduzierung unnötiger Informationen verringert werden. Es gibt unterschiedliche Reduzierungstechniken, die für unterschiedliche Ressourcentypen angewendet werden können (Kuhn/Raith 2013, 218).
— Verwendung protokollspezifischer Performance – Optimierungstechniken (Grigorik 2013, 236).

Die Funktionsweisen der verschiedenen Versionen des HTTP – Protokolls unterscheiden sich sehr. Aus diesem Grund muss genau beachtet werden, welche Performance – Optimierungstechniken für welches HTTP – Protokoll geeignet sein können (Grigorik 2013, 236). Manche der oben beschriebenen Techniken, die sich unabhängig von der Version des HTTP – Protokolls anwenden lassen, werden im Folgenden genauer dargestellt.

3.2.1. Browser Caching für statische Ressourcen

Um zusätzliche Bytes während des Transports sparen zu können, kann der Cache des Browsers verwendet werden. Die Caching – Anweisungen werden durch Metainformationen im HTTP – Header der HTTP – Nachricht übertragen. Dadurch werden die betroffenen Daten über längere Zeit im Cache des Browsers gespeichert. Es gibt mehrere Möglichkeiten, mithilfe derer die Caching – Anweisungen dem Client mitgeteilt werden können (Kuhn/Raith 2013, 168).

Die erste Möglichkeit ist der Einsatz von „Expires – Header". Dazu müssen die HTTP – Header der einzelnen Responses ein „Verfallsdatum" beinhalten. Danach wird der Browser dieses „Verfallsdatum" interpretieren und mit der dazugehörigen Datei bis zum angegebenen Zeitpunkt speichern. „Expires – Header" sind Caching – Anweisungen, die für jede Response angewendet werden können. Die Dateien werden im Browser Cache so lange gehalten, bis das „Verfallsdatum" erreicht wird. Um die „Expires – Header" nutzen zu können, müssen diese Serverseitig aktiviert und je nach Ressourcentyp konfiguriert werden (Kuhn/Raith 2013, 168).

Der Nachteil von „Expires – Headern" besteht darin, dass diese sich mit einer Änderung der Datei nicht dynamisch anpassen werden. Dies bedeutet, dass die Datei so lange vom Browser Cache aufgerufen wird, bis das „Verfallsdatum" erreicht ist, auch dann wenn die Datei serverseitig aktualisiert wurde (Kuhn/Raith 2013, 170).

Eine zweite Möglichkeit ist es, um die sog. „Etags" zu benutzen. Diese Technologie ist den „Expires – Headern" ähnlich, aber hier werden die HTTP – Header des Responses mit einem „Etag" markiert. Ein „Etag" wird im Request vom Browser mithilfe von *„If-None-Match: (Etag)."* gesetzt. Am Server wird ein neuer „Etag" für die angeforderte Datei erzeugt und dieser wird mit dem vom Client gesendeten „Etag" verglichen. Wenn beide „Etags" nicht identisch sind, bedeutet es, dass der Client eine ältere Version der Datei hat. In diesem Fall wird der Server eine neue Version der angefragten Ressource schicken. Wenn die „Etags" identisch sind, wird der Webserver mit dem Statuscode „304 – Not Modified" antworten und keine Inhalte zurückliefern (Kuhn/Raith 2013, 171).

Der Vorteil von „Etags" im Vergleich zu „Expires – Headern" besteht darin, dass Dateiänderungen erkannt werden. Jedoch werden in diesem Fall keine Requests gespart. Die „Etags" sind in einigen Webservern z.B. Apache2 (Apache Software Foundation 2016d) und Lighttpd (Lighttpd 2016) in der Standardkonfiguration aktiviert (Kuhn/Raith 2013, 171).

Eine dritte Möglichkeit sind die „Cache – Buster". Diese Technologie wird entweder durch die Ergänzung von Parametern an den Dateinamen oder durch die Veränderung des Dateipfades realisiert. Der Browser identifiziert die zwischengespeicherte Datei mithilfe des dazugehörigen Dateipfads. Wenn dieser mithilfe von zusätzlich angehängten Parametern an den Dateinamen (z.B wird aus „ressourcen/code.js" „ressourcen/ code.js?v=2") geändert wird, wird die Ressource als neu betrachtet und neu geladen (Kuhn/Raith 2013, 173).

Der Nachteil der oben beschriebenen Technologie liegt darin, dass durch den Proxyserver diese Ergänzungen abgeschnitten und verworfen werden können. Dadurch wird die Anfrage nicht mehr korrekt sein (Kuhn/Raith 2013, 173).

Die zweite Variante des „Cache – Buster" besteht in der Änderung des kompletten Dateipfades, der automatisch umgeschrieben wird. Z.B. kann der Pfad anstatt der ursprünglichen Anfrage „ressourcen/code.js" dann „ressourcen/123/code.js" lauten. Diese automatische Umschreibung kann auf der Serverseite mithilfe von „Rewrite" – Regeln realisiert werden (Kuhn/Raith 2013, 173).

3.2.2. Datenkompression während des Transports

Die „Gzip" – Kompression ermöglicht es, Daten während des Transports vom Server zum Client zu komprimieren. Bei den quelltext-basierten Ressourcen, wie HTML, CSS, JavaScript u.s.w. können mithilfe der „Gzip" – Kompression während des Transports etwa 60-80% der Datenmenge gespart werden (Grigorik 2013, 237). Bei schon komprimierten Formaten (JPG, MP3 u.s.w) ist eine zusätzliche „Gzip" – Kompression nutzlos (Wikipedia 2016f). Wenn die „Gzip" – Kompression aktiviert ist, können vom Server gesendete Nachrichtenrümpfe (HTTP – Body) komprimiert werden. HTTP – Header werden innerhalb des HTTP/1.1 – Protokolls nicht komprimiert. Um die „Gzip" – Kompression verwenden zu können, muss der Webserver entsprechend konfiguriert werden (Kuhn/Raith 2013, 211).

Für binäre Daten (z.B. Bilder) wird eine andere Kompressionstaktik angewendet. Später wird noch beschrieben, wie man Quelltext-basierte Ressourcen minimieren kann.

3.2.3. Reduzierung der Datenmenge der Ressourcen

Die Datengröße der quelltextbasierten Daten kann mithilfe von Minimierungen reduziert werden. Textbasierte Quellcodes wie z.B. HTML, JavaScript oder CSS Dateien verfügen oftmals über Tabulatoren, Kommentare, Zeilenumbrüche u.s.w. Diese sind für die Maschinenverarbeitung nicht wichtig und können entfernt werden, weil diese die Dateigröße erhöhen. Außerdem gibt es datenspezifische Minimierungstechniken, die je nach Ressourcentyp angewendet werden

können. Z.B. gibt innerhalb der HTML Datei „Tags", die kein schließendes „Tag" benötigen. Wenn diese entfernt werden, wird weniger Bandbreite für die Dateiübertragung benötigt (Kuhn/Raith 2013, 215).

Es gibt noch weitere datenspezifische Minimierungstechniken. Innerhalb des JavaScript Codes können die Variablen- und Methodennamen verkürzt werden. So entstehen z.B. anstatt Funktionsnamen wie „function1" einfach Namen wie „f". Die meisten JavaScript-Bibliotheken verfügen über bereits minimierte Versionen des Quellcodes. Diese haben am Dateinamen den Anhang „min" (Kuhn/Raith 2013, 216).

CSS Dateien können auch zusätzlich spezifisch minimiert werden. Normalerweise verfügen die CSS – Selektoren über mehrere CSS – Anweisungen. Manche CSS – Anweisungen können in einem Befehl zusammengefasst werden. Dies betrifft z.B. „margin", „padding", „border" und „background" (Kuhn/Raith 2013, 218).

Nachdem die Daten minimiert werden, werden diese für den Entwickler schlechter lesbar sein. Deshalb ist es empfehlenswert, um die Daten für Entwicklung und Veröffentlichung separat zu halten.

Für die Minimierung gibt es zahlreiche Online Tools, die sowohl allgemeine als auch datenspezifische Techniken anwenden können. Als Alternative kann ein automatisierendes Build – System wie „Gulp" (Fractal 2016) oder „Grunt" (Grunt Development Team 2016) in den Entwicklungsprozess eingebunden werden. Diese Tools können oben beschriebene Minimierungstechniken anwenden.

3.2.4. Bildoptimierung

Etwa 60% einer durchschnittlichen Internetseite machen Bilder aus. Mittels Bildoptimierung kann die Gesamtladezeit der Webapplikation deutlich reduziert werden (Kuhn/Raith 2013, 219). Dafür gibt es mehrere Techniken, die nach Kuhn/Raith (2013, 219) aufgelistet werden:

- Wahl der richtigen Art der Computergrafik (Vektorgrafiken vs. Rastergrafiken) und des richtigen Dateiformats.
- Richtige Optimierungstechniken bei den ausgewählten Bildformaten beachten.
- Entfernung von Metadaten.
- Anstatt Bilder CSS – Effekte verwenden, keinen Text in die Bilder schreiben.

Wahl der richtigen Art der Computergrafik (Vektorgrafiken vs. Rastergrafiken) und des richtigen Dateiformats.

Die Auswahl des richtigen Bildformats ist für die korrekte Darstellung des Bildes sehr wichtig. Hauptsächlich unterscheiden sich zwei Arten von Computergrafiken: Vektorgrafiken und Ras-

tergrafiken. Die Rastergrafiken dienen der Darstellung von komplizierten Bildern mit vielen Details und unregelmäßigen Farben. Dabei muss beachtet werden, wie gut bei unterschiedlichen Zoomstufen die Bilder dargestellt werden (Grigorik 2016b).

Die Vektorformate sind vom Zoomfaktor und der Auflösung unabhängig. Deshalb ist es empfehlenswert, in Webapplikationen Vektorgrafiken vorzuziehen, weil diese sich unabhängig von der Auflösung und der Größe für die Darstellung auf vielen Endgeräten eignen (Grigorik 2016b).

Durch die richtige Wahl des Bildformates kann die Webapplikation deutlich optimiert werden. Die Beschreibung des einzelnen Bildformates überschreitet jedoch den Rahmen dieser Masterarbeit. Das wichtigste ist, dass die Bilder eine möglichst kleine Dateigröße haben und immer noch mit akzeptabler Qualität dargestellt werden. Aus diesem Grund sollte man immer einschätzen, welches Bildformat zu einem Bild am besten passen könnte.

Richtige Optimierungstechniken bei den ausgewählten Bildformaten beachten.

Für alle Bildformate existieren sowohl verlustfreie als auch verlustbehaftete Komprimierungen. Bei verlustbehafteter Komprimierung wird ein Filter angewendet, der vom Bild einige Pixeldaten entfernt. Dabei muss man beachten, dass sich bei jedem einzelnen Bildformat das verlustbehaftete Komprimierungsverfahren unterscheidet. Bei verlustfreier Komprimierung wird ein Filter angewendet, der die Pixeldaten komprimiert. Alle Bildformate haben eine eigene Kombination von Algorithmen, die die Funktionsweise der Komprimierung bestimmen. Um die passende Optimierungstechnik für jedes einzelne Bildformat zu finden, muss die Bearbeitungsweise jedes Bildformates genau betrachtet werden (Grigorik 2016b).

Entfernung von Metadaten.

Die Bilder können über Zusatzinformationen (sog. Metadaten) verfügen. Diese können Kommentare, Anwendungs- oder EXIF (Exchangeable Image File Format) – Informationen sein. Diese enthalten z.B. Angaben zum Kameratyp, dem Aufnahmedatum, Kommentare usw. Es können auch Thumbnails für eine schnellere Vorschau des Bildes gespeichert werden (Kuhn/Raith 2013, 221). Diese Informationen sind für die Darstellung des Bildes in der Webapplikation nicht relevant und bestimmen die Bildqualität nicht. Die Metainformationen können zusätzlichen Speicherplatz von bis zu 30Kb pro Bild haben. Diese Informationen sollten entfernt werden (Kuhn/Raith 2013, 221).

Bei der Entfernung der Metadaten können sowohl viele Bildverarbeitungsprogramme als auch Plugins für die Bildoptimierung in den Taskmanagern (wie „Gulp" oder „Grunt") helfen.

Anstatt Bilder CSS – Effekte verwenden, keinen Text in Bilder schreiben.

Man sollte immer auf die Konstruktion und die Bestandteile des Bildes achten. Bei vielen kleinen Bildern (wie einfache Symbole, Pfeile oder einfache geometrische Figuren) kann man die

Rastergrafiken durch CSS – Effekte oder Webschriften ersetzen. CSS – Effekte haben mehrere Vorteile gegenüber Bildern (Grigorik 2016b):

- Sie werden bei jeder Auflösung und Zoomstufe immer scharf bleiben.
- Das „Gewicht" der Seite wird reduziert.
- Es werden weniger HTTP – Requests stattfinden.

(Grigorik 2016b).

Außerdem darf man die Schrift, die zu einem Bild gehört, nicht außer Acht lassen. Es ist immer besser, Text nicht direkt ins Bild zu schreiben, sondern Webschriften zu benutzen. Es gibt mehrere Vorteile von Webschriften (Grigorik 2016b):

- Man kann die Größe der Schriften oder die Schriftart immer verändern, wenn sich das Design der Webapplikation ändern wird. Außerdem kann man unabhängig von dem Bild arbeiten.
- Es gibt sehr viele Schriftarten, die auch von den Browsern unterstützt werden.

(Grigorik 2016b).

3.3. Vorstellung des HTTP/2 – Protokolls

HTTP/2 ist der Nachfolger des früher entwickelten SPDY – Protokolls. Die Spezifikation des SPDY – Protokolls wurde als Startpunkt für das neue HTTP/2 – Protokoll adaptiert. Das SPDY – Protokoll ist ein experimentelles Protokoll, das von „Google" Mitte des Jahres 2009 entwickelt wurde. Das wichtigste Ziel, das dieses Protokoll erreichen sollte, war die Überwindung der gut bekannten Einschränkungen des HTTP/1.1 – Protokolls. Nach Grigorik (2015, 2-3) sind die Ziele:

- Gesamtladezeit um 50% reduzieren.
- Bei der Benutzung des neuen Protokolls Änderungen für die Endanwender vermeiden.
- Bedarf an Änderungen am Inhalt der Webapplikation für die Webadministratoren vermeiden.
- Entwicklungskomplexität reduzieren.
- Entwicklung im Rahmen von Open-source.
- Reelle Performance – Daten für die Validierung des neuen Protokolls erfassen.

(Grigorik 2015, 2-3).

Das SPDY – Protokoll ist ein binäres Protokoll, das die Benutzung der TCP – Verbindung durch mehrere Funktionen effizienter machen kann. Dazu gehören: Multiplexierung von Requests und Responses, Priorisierungen und Kompression der Headerfelder (Grigorik 2015, 3).

Das HTTP/2 – Protokoll ist ein neues Internetprotokoll, das im Februar 2015 von der IESG (The Internet Engineering Steering Group) genehmigt wurde. Im Mai des gleichen Jahres ist die offizielle Dokumentation des Protokolls erschienen: RFC – 7540 für das HTTP/2 – Protokoll (Belshe et al. 2015) und RFC – 7541 für die Header Kompression für HTTP/2 (Peon/Ruellan 2015; Grigorik 2015, 4). Das neue Protokoll ermöglicht mithilfe der Kompression von Headerfeldern und der Multiplexierung, die innerhalb einer TCP – Verbindung stattfindet, eine effizientere Nutzung der Netzwerkressourcen und verringert die Latenz (Belshe et al. 2015).

Das Ziel des HTTP/2 – Protokolls ist einerseits die Überwindung der gut bekannten Schwachstellen und Begrenzungen des HTTP/1.1 – Protokolls und anderseits die Erweiterung des HTTP/1.1 – Protokolls durch zusätzliche Funktionen:

- Kompression der HTTP - Headerfelder während dem Transport zwischen Client und Server.
- Bewältigung des „Head-Of-Line Blocking" Problems in der HTTP – Schicht.
- Multiplexierung der Requests und Responses.
- Genaue Definition, wie das HTTP/2 – Protokoll mit dem HTTP/1.1 – Protokoll interagieren wird.
- Priorisierung der Requests und Responses.
- Anwendung des „Server Pushs".

(Grigorik 2013, 209-210).

Das HTTP/2 – Protokoll nutzt die gleichen Semantiken, wie das HTTP/1.1 – Protokoll. Es wurden keine großen Änderungen in den HTTP – Methoden, Statuscodes, URIs und Headerfeldern durchgeführt. Große Änderungen haben vor allem die Übertragungsart der Daten zwischen Client und Server betroffen (Grigorik 2013, 210).

Das neue Protokoll wird von den meisten verwendeten Browsern unterstützt (Abb. 9).

Abb. 9: Browserunterstützung des HTTP/2 – Protokolls, Stand: 26.08.2016 (<http://caniuse.com/#search=http2>).

Die serverseitige Unterstützung des HTTP/2 – Protokolls und die Serverwahl wird im Teil „Testvorbereitung" beschrieben.

3.3.1. Wichtigste Bestandteile des HTTP/2 – Protokolls

Im Vergleich zum HTTP/1.1 – Protokoll, in dem die Daten im Klartext transportiert wurden, hat das HTTP/2 – Protokoll ein binäres Format. Die Protokolle, die die ASCII – Kodierung implementieren, sind meistens einfach zu untersuchen und zu benutzen. Allerdings sind diese nicht effizient genug und es ist meistens schwieriger, um diese korrekt zu implementieren. Durch zusätzliche Leerzeichen, ändernde Abbrüche und weitere Eigenschaften ist es schwer, das Protokoll unter Nutzdaten, Parserhinweisen und Sicherheitsfehlern zu erkennen. Im Gegenteil zu Protokollen mit ASCII – Kodierung ist die Benutzung des binären Protokolls schwieriger, aber diese neigen dazu perfomanter, stabiler und mit der korrekteren Implementierung versehen zu sein (Grigorik 2015, 7).

In Abb. 10 ist die binäre Frameschicht des HTTP/2 – Protokolls dargestellt. Diese schreibt vor, wie HTTP – Nachrichten zwischen dem Client und dem Server eingekapselt und transportiert werden müssen (Grigorik 2013, 211).

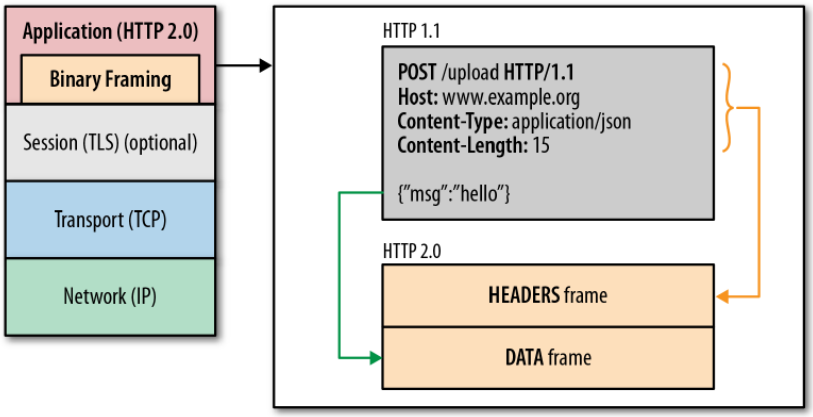

Abb. 10: Binäre Frameschicht des HTTP/2 - Protokolls (Grigorik 2013, 212).

Es gibt eine wichtige Voraussetzung für die Nutzung des HTTP/2 – Protokolls: sowohl Client als auch Server müssen einen Mechanismus der binären Kodierung unterstützen, damit sich beide Seiten der Kommunikation verstehen werden (Grigorik 2015, 6).

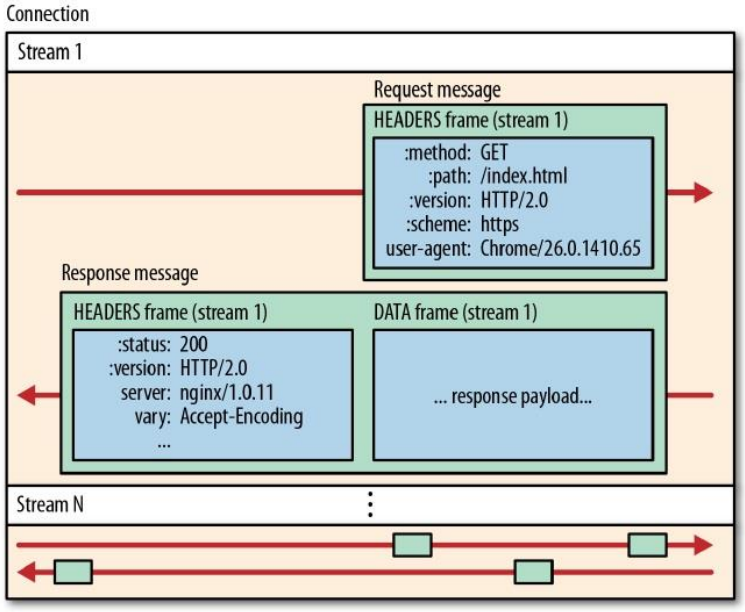

Abb. 11: „Streams", „Messages" und „Frames" unter dem HTTP/2 – Protokoll (Grigorik 2013, 213).

Ein Grundkonzept von Streams, Messages und Frames.

Für die Kommunikation zwischen Client und Server wird nur eine TCP – Verbindung benötigt. Die Kommunikation unter dem HTTP/2 – Protokoll wurde in „Streams", „Messages" und „Frames" aufgeteilt (Abb. 11).

„Stream" ist ein Bytestrom innerhalb der bestehenden Verbindung. Innerhalb eines „Streams" werden eine oder mehrere „Messages" transportiert (Grigorik 2015, 7). Die Anzahl an „Streams", die gleichzeitig bearbeitet werden kann, wird per Server definiert. Jeder „Stream" hat einen einzigartigen Identifikator (ID), der nicht wiederverwendbar ist.

Eine „Message" ist eine logische HTTP – Nachricht, die aus Requests und Responses besteht. Diese kann einen oder mehrere „Frames" in sich haben (Grigorik 2015, 7 - 8).

„Frame" ist die kleinste Maßeinheit innerhalb der Kommunikation unter dem HTTP/2 – Protokoll. Jeder „Frame" verfügt über einen „Frame Header", der zumindest Informationen über den Stream beinhaltet, zu dem dieser gehört. „Frames" definieren den spezifischen Typ von Daten, wie z.B. HEADERS – Frame für den Nachrichtenkopf (HTTP – Header), DATA – Frame für den Nachrichtenrumpf (HTTP – Nutzdaten) u.s.w. Alle „Frames" werden im binären Format dargestellt (Grigorik 2015, 7 - 8).

Es gibt mehrere Typen von „Frames“, die im HTTP/2 – Protokoll spezifiziert sind (Tabelle 2).

TYP	Frame	Beschreibung
0	DATA	Wird für den Transport der Nutzdaten der HTTP – Nachrichten (Body) benutzt.
1	HEADERS	Wird für die Eröffnung eines „Streams“ und die Übertragung der HTTP – Headerfelder eines „Streams“ verwendet.
2	PRIORITY	Spezifiziert die Priorität eines „Streams“, die vom Sender angekündigt ist.
3	RST_STREAM	RST_STREAM wird für die sofortige Beendigung eines „Streams“ gesendet, oder um zu signalisieren, dass ein Fehler passiert ist.
4	SETTINGS	Wird für die Übertragung von Konfigurationsparametern benutzt, um zu bestimmen, wie die Kommunikationspartner miteinander interagieren werden.
5	PUSH_PROMISE	Kündigt dem Client einen Stream vom Server an, mit den im primären Stream referenzierten Ressourcen.
6	PING	Wird sowohl für die Messung des minimalen „Round Trip Time“ vom Absender benutzt, als auch für die Bestätigung, dass die bestehende Verbindung noch funktionsfähig ist. Diese „Frames“ können von jedem Kommunikationspartner gesendet werden.
7	GOAWAY	Wird für die Beendigung der bestehenden Verbindung oder zum Signalisieren massiver Fehlermeldungen verwendet. Dieser Frametyp erlaubt es, die Erzeugung von neuen „Streams“ zu unterbinden, während früher erzeugte „Streams“ fertig bearbeitet werden.
8	WINDOW_UPDATE	Wird für die Implementierung von „Flow Control“ benutzt.
9	CONTINUATION	Wird für die Fortsetzung einer Sequenz von Header Block - Fragmenten (ein Bestandteil des „Frames“: HEADERS, PUSH_PROMISE und CONTINUATION) verwendet. Es kann eine beliebige Anzahl von diesen „Frames“ gesendet werden, wenn der früher gesendete „Frame“ zu den Typen HEADERS, PUSH_PROMISE oder CONTINUATION gehört, die keinen „END_HEADERS“ - Flag an sich haben und zum gleichen „Stream“ gehören.

Tab. 2: Typen von HTTP/2 „Frames“ (Belshe et al. 2015, 31-49)

Nachdem die Verbindung des HTTP/2 – Protokolls aufgebaut ist, können Server und Client anfangen, um die „Frames“ auszutauschen. Alle „Frames“ haben einen auf 9 Byte fixierten „Frame Header“, auf den die Nutzdaten („Frame Payload“) folgen. Der „Frame Header“ hat mehrere

Felder: Length, Type, Flags und Stream Identifier (Abb. 12) „Frame Payload " hat keine fixierte Länge: sie unterscheidet sich je nach Typ des „Frames" (Belshe et al. 2015, 12).

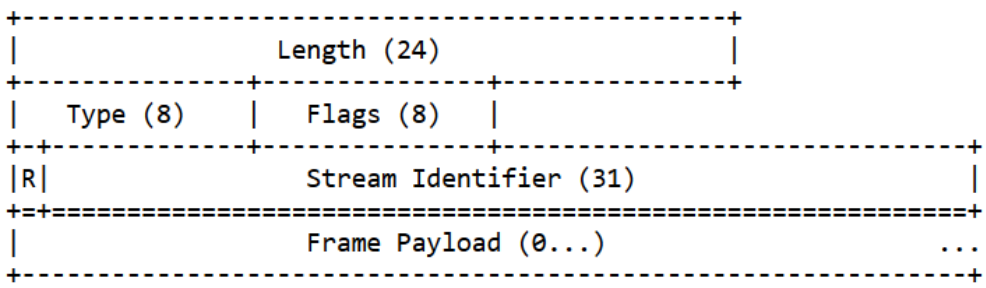

Abb. 12: Bestandteile eines „Frames" (Belshe et al. 2015, 12).

Length Feld

Das „Length Feld" definiert die maximale Größe der Nutzdaten und ist vom Empfänger innerhalb der SETTINGS_MAX_FRAME_SIZE in den Einstellungen des HTTP/2 – Protokolls definiert. Diese kann einen Wert zwischen 214 Bit und 224 Bit haben. Der „Frame Header" ist in diesen Daten nicht enthalten. Gleichzeitig wird im HTTP/2 – Standard die maximale Größe auf 2^{24} - 1 Bit gesetzt, um Client und Server höhere Werte zum Senden zu erlauben. Je kleiner der Wert während der Übertragung gehalten wird, desto effizienter funktioniert die Multiplexierung (Belshe et al. 2015, 12-13).

Type Feld

„Type" ist ein 8 - Bit Feld. In diesem Feld wird der Typ des „Frames" (z.B. HEADERS, DATA, u.s.w.) und seine Semantiken dargestellt. Der Frame „Type" bestimmt in Bezug auf die HTTP/2 Implementierung sowohl das Format als auch die Semantik eines Frames (Belshe et al. 2015, 12).

Flags Feld

„Flag" ist ein 8 - Bit Feld, das für framespezifische boolesche „Flags" reserviert ist. Flags werden Semantiken zugewiesen, die spezifisch für den entsprechenden Frame-Typ sind (Belshe et al. 2015, 12).

R Feld

Das „R Feld" ist ein Feld, das immer auf 1 Bit reserviert ist. Die Semantiken dieses Bits sind undefiniert. Dieses Feld wird immer auf den Wert 0 gesetzt (Belshe et al. 2015, 12).

Stream Identifier Feld

Das „Stream Identifier" Feld ist ein 31 - Bit integer Feld, das den „Stream" einzigartig identifiziert (Belshe et al. 2015, 12).

Eröffnung und Beendigung eines Streams

Bevor die Daten der Webapplikation innerhalb des HTTP/2 – Protokolls gesendet werden können, muss ein neuer „Stream" geöffnet werden. Ein „Stream" kann sowohl clientseitig als auch serverseitig geöffnet werden: der Client kann mithilfe des HEADERS – Frame und der Server kann mithilfe des PUSH_PROMISE – Frames einen neuen „Stream" öffnen. Um Kollisionen zwischen clientseitigen und serverseitigen „Streams" zu vermeiden, haben diese unterschiedliche IDs. Serverseitige „Streams" werden mit geraden IDs und clientseitige werden mit ungeraderen IDs gekennzeichnet (Grigorik 2015, 26 - 27), (Belshe et al. 2015, 40-42).

Um geöffnete „Streams" zu beenden, wird ein RST_STREAM – Frame gesendet. Nachdem ein GOAWAY – Frame gesendet wird, werden keine „Streams" innerhalb der bestehenden TCP – Verbindung mehr erzeugt (Belshe et al. 2015, 36, 43).

3.3.2. Request und Response Multiplexierung

Unter dem HTTP/1.1 – Protokoll kann nur eine Antwort vom Server (Response) innerhalb einer TCP – Verbindung geliefert werden. Aus diesem Grund entsteht eine Warteschlange von Responses, die die Bezeichnung „Head-Of-Line Blocking" hat. Dies macht die Nutzung der TCP – Verbindung sehr ineffektiv (Grigorik 2015, 9).

Im Vergleich zum HTTP/1.1 – Protokoll, werden im HTTP/2 – Protokoll die Requests und Responses in einzelne „Frames" aufgeteilt, die innerhalb eines „Streams" transportiert werden. Innerhalb des HTTP/2 – Protokolls ist es möglich geworden, um die Lieferung einzelner „Frames" von Requests und Responses zu mischen. Vor dem Versand werden HTTP – Nachrichten vom Client und Server auf einzelne „Frames" aufgeteilt und nach dem Empfang auf der anderen Seite wieder zusammengesetzt (Grigorik 2015, 9). Abb. 13 veranschaulicht diesen Prozess.

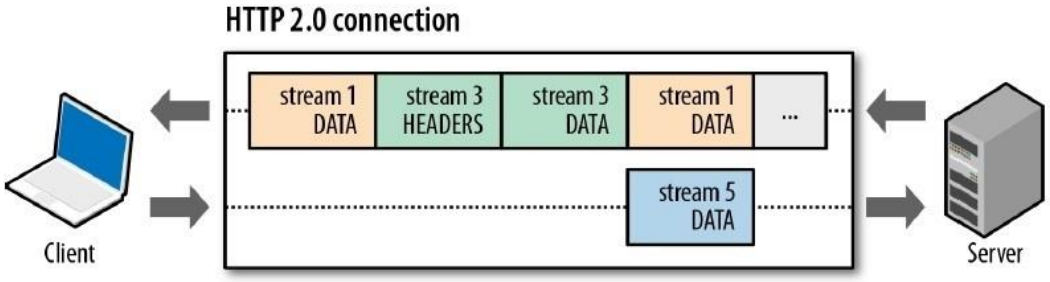

Abb. 13: Request und Response Multiplexierung innerhalb einer TCP Verbindung (Grigorik 2015, 9).

Mithilfe dieser Besonderheit des neuen Protokolls soll die Performance der Webapplikationen deutlich besser sein:

- Sowohl Requests als auch Responses werden multiplexiert und blockieren sich nicht mehr gegenseitig.
- Es wird nur eine TCP – Verbindung benötigt, um die wechselnden Requests und Responses gleichzeitig in beide Richtungen liefern zu können.
- Die HTTP/1.1 – Optimierungstechniken, wie Sprites von Bildern, Zusammenfügen der Dateien (JavaScript, CSS) und Aufteilung der Ressourcen auf unterschiedliche Domains werden nicht mehr benötigt.
- Dadurch, dass im HTTP/2 – Protokoll das „Head-Of-Line Blocking" Problem auf der HTTP – Ebene besser gelöst ist, wird die Kapazität des Netzwerks effizienter genutzt.

(Grigorik 2015, 9 - 10).

3.3.3. Stream Priorisierung

Aufgrund der Besonderheit des HTTP/2 – Protokolls, HTTP – Nachrichten auf einzelne „Frames" aufteilen zu können, öffnen sich zusätzliche Möglichkeiten. Außer dass „Frames" unterschiedlicher „Streams" unabhängig voneinander geliefert werden können, kann die Reihenfolge, in der die „Frames" am Client ankommen, eine große Rolle für den kritischen Rendering – Pfad spielen. Um dies zu ermöglichen, wurde dem HTTP/2 – Protokoll noch eine weitere Funktion hinzugefügt: „Stream Priorisierung" (Grigorik 2015, 10).

Die Spezifikation des HTTP/2 – Protokolls erlaubt es, jedem „Stream" Abhängigkeiten und Gewichte zu geben:

- Jeder „Stream" darf ein Gewicht (ein Integer-Wert) zwischen 1 und 256 haben.
- Jeder „Stream" kann in Abhängigkeit mit einem anderen „Stream" stehen (Grigorik 2015, 10).

Die Kombination zwischen Abhängigkeiten und Gewichten eines jeden „Streams" erlaubt dem Client einen Priorisierungsbaum zu bauen. Der Priorisierungsbaum ist ein Hinweis für den Server, in welcher Reihenfolge die angefragten Ressourcen am besten geliefert werden sollen. Der Server kontrolliert gleichzeitig die Auslastung von CPU, RAM und Bandbreite und soweit die Daten zum Senden zur Verfügung stehen, werden diese gemäß den vom Client gelieferten Prioritäten an den Client geliefert. Allerdings kann der Client keine genaue Lieferreihenfolge vom Server verlangen. Es ist außerdem nicht gewünscht, die Serververarbeitung zu verlangsamen, wenn Ressourcen mit höherer Priorität blockiert sind (Grigorik 2015, 10-12). Im schlimmsten Fall kann das „Head-Of-Line Blocking" - Problem zurückkehren.

Alle „Streams" im HTTP/2 – Protokoll hängen von einem „Root Stream" ab. Abhängigkeit eines „Streams" im HTTP/2 – Protokoll bedeutet, dass der „Eltern Stream" vor seinen „Kinder Streams" bearbeitet werden soll. In Abb. 14, Beispiel B, wird gezeigt, dass „Stream C" von

„Stream D“ abhängig ist. Dies bedeutet, dass „Stream D“ alle verfügbaren Bandbreiteressourcen bekommt und vom Server als erster verarbeitet wird (Grigorik 2015, 11). Um die Abhängigkeiten zu zeigen, verweisen die „Kinder Streams“ mithilfe einer „Parent Stream ID“ auf ihre „Eltern Streams“.

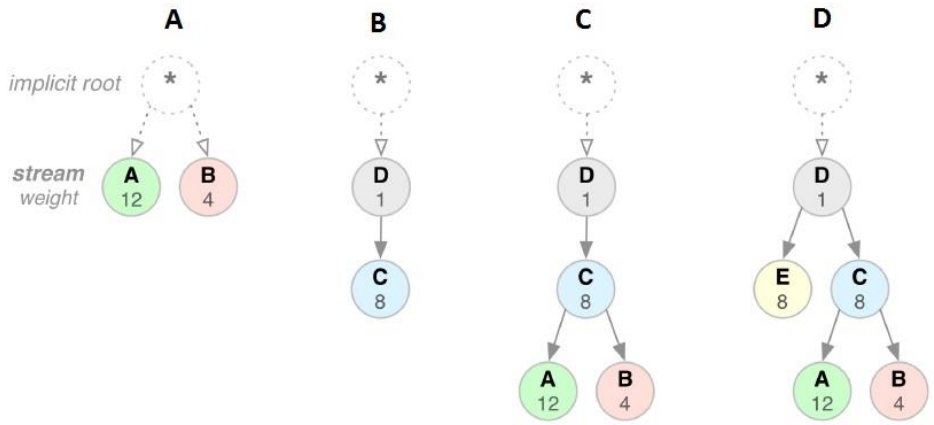

Abb. 14: Abhängigkeiten und Gewichte von Streams (Grigorik 2015, 11).

Zwischen „Streams“ mit gemeinsamem „Eltern Stream“, wird die Bandbreite je nach zugewiesenem Gewicht eines jeden „Streams“ aufgeteilt. Je größer das Gewicht eines „Streams“ ist, desto mehr verfügbare Bandbreiteressourcen bekommt er. In Abb. 14 sind mehrere Beispiele dargestellt. Die Aufteilung der Bandbreite wird für jedes Beispiel kurz beschrieben.

Im Beispiel A haben „Stream A“ und „Stream B“ einen gemeinsamen „Root - Stream“. Da „Stream A“ das Gewicht 12 hat und „Stream B“ ein Gewicht von 4 hat, wird „Stream A“ 2/3 aller verfügbaren Bandbreiteressourcen bekommen (Grigorik 2015, 11).

Im Beispiel B ist „Stream C“ von „Stream D“ abhängig. Deshalb wird „Stream D“ zuerst alle verfügbaren Bandbreiteressourcen bekommen (Grigorik 2015, 11).

Im Beispiel C wird zuerst „Stream D“ die gesamte Bandbreite bekommen, danach „Stream C“. Anschließend wird die Bandbreite zwischen „Stream A“ und „Stream B“ aufgeteilt. Nur wird „Stream A“ 2/3 aller verfügbaren Bandbreiteressourcen bekommen (Grigorik 2015, 11).

Im Beispiel D wird zuerst „Stream D“ alle verfügbaren Bandbreiteressourcen bekommen. Danach wird die Bandbreite zwischen „Stream E“ und „Stream C“ gleichmäßig aufgeteilt. Am Ende wird die Bandbreite je nach den Gewichten zwischen „Stream A“ und „Stream B“ aufgeteilt (Grigorik 2015, 11).

In der HTTP/2 – Spezifikation ist es erlaubt, dass der Client Abhängigkeiten und Gewichte zu jedem Zeitpunkt aktualisiert (Grigorik 2015, 12). Priorisierungen von „Streams“ sind nur Hinweise und deren Implementierungsstrategie hängen sowohl vom Server, als auch vom Client ab: Der Client muss geeignete Priorisierungsdaten erstellen und der Server muss diese Daten

annehmen und in der entsprechenden Priorisierungsreihenfolge zurück an den Client liefern können (Buch IG, 216).

Es kann jedoch vorkommen, dass der Server die Priorisierungsinformationen nicht akzeptiert (Buch IG, 216). Dadurch können Hindernisse für den kritischen Rendering – Pfad entstehen: anstatt kritische Ressourcen zuerst zu liefern, wird der Server nicht kritische Ressourcen senden.

Außerdem unterstützen nicht alle modernen Browser die Priorisierungen (Tabelle 3) (Ishizawa 2015, 7).

Namen	Version	Unterstützung
Mozilla Firefox	38.0.5	Ja
Google Chrome	43.0.2357.65	Nur Gewichte
Microsoft Edge	0.11.10074.0 (Preview).	Nein
Apple Safari	9,0 (Preview).	Nein

Tab. 3: Implementierung der HTTP/2 – Priorisierungen im Browser (Ishizawa 2015, 7).

Es ist schwer, um genau vorherzusagen, wie die Priorisierungsbäume von unterschiedlichen Browsern gebaut werden. Der Grund dafür ist, dass jeder Browser auf seine eigene Art die Priorisierungen erstellt. In „Mozilla Firefox" liegen die Prioritäten zuerst bei CSS Dateien, danach bei JavaScript Dateien und zuletzt bei den Bildern. „Google Chrome" erstellt Priorisierungen nur auf Basis von Gewichten. Dies ist allerdings nicht effizient genug. „Microsoft Edge" unterstützt Priorisierungen nicht (Moto Ishizawa, „Understanding HTTP/2 prioritization", Folie 64). Aus diesem Grund ist es schwer, um die clientseitige Priorisierungsimplementierung unter Kontrolle zu bringen.

Prioritäten für „Streams" werden mithilfe des „PRIORITY" – Frames vom Sender angekündigt. Dieser „Frame" hat innerhalb der Nutzdaten („Frame" – Payload) nur zwei Felder: „Stream Dependency" – für die Abhängigkeit und „Weight" – für das Gewicht (Belshe et al. 2015, 34-35).

3.3.4. Header Kompression

„Header Kompression" (kurz „HPACK") ist ein Kompressionsformat für die effizientere Übertragung von HTTP – Headerfeldern innerhalb der HTTP/2 – Verbindung (Peon/Ruellan 2015, 4).

Innerhalb des HTTP/1.1 – Protokolls wurden alle Request - und Response Headerfelder ohne Kompression im Klartext übertragen. Wenn eine Webapplikation innerhalb einer Sitzung mehrere Dutzend Requests braucht, wächst der Anteil der angefragten Header zunehmend. Dies verbraucht Bandbreite und führt dazu, dass zusätzliche Latenz entsteht (Peon/Ruellan 2015, 4). Manchmal kann diese Größe bis zu 800 Bytes (ohne Cookies) pro Request - Response erreichen (Grigorik 2015, 19).

Viele Metadaten, die als Schlüssel – Werte Paare dargestellt sind, unterscheiden sich von Request zu Request oder von Response zu Response oftmals nicht voneinander. Dazu gehören

z.B. Method: GET, User - Agent: Mozilla, Chrome, u.s.w. Diese wiederholen sich und verbrauchen die Bandbreiteressourcen. Aus diesem Grund werden in „HPACK" sowohl statische als auch dynamische Tabellen verwendet (Abb. 15). Eine Statische Tabelle ist in der Spezifikation definiert und besteht aus den üblichen HTTP – Headerfeldern, die in allen Requests und Responses benutzt werden. Jedes Headerfeld hat in der statischen Tabelle seine eigene Nummer (Peon/Ruellan 2015, 4), (Grigorik 2015, 19).

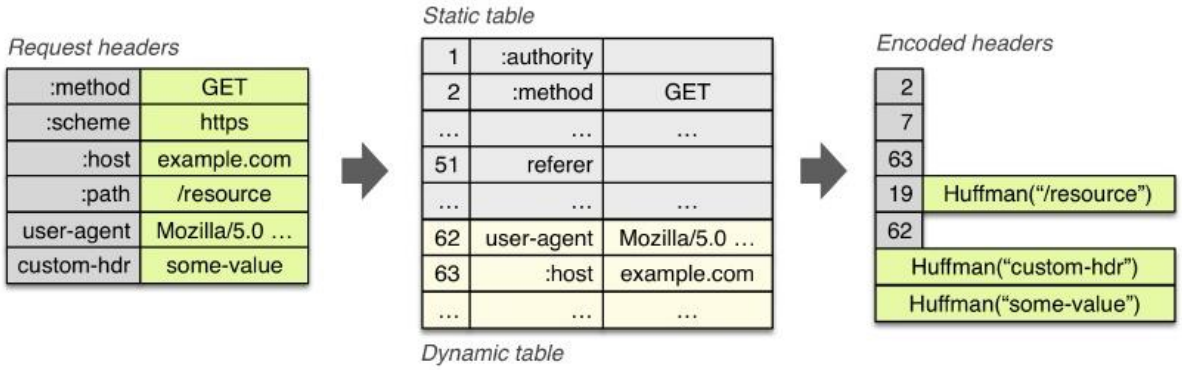

Abb. 15: Funktionsweise des „HPACK" in HTTP/2, (Grigorik 2015, 19).

Außer den üblichen Werten existieren auch andere Schlüssel - Werte Paare in Headerfeldern, die zur bestimmten Webapplikation gehören (z.B. Servername) und sich mit zunehmenden Anfragen und Antworten mehrmals wiederholen können. Dafür wird die dynamische Tabelle angewendet. Zuerst ist die dynamische Tabelle leer, wird sich aber innerhalb der HTTP/2 – Verbindung erweitern. Jeder Wert in der dynamischen Tabelle wird auch indexiert (Abb. 15). Anstatt ganze Schlüssel – Werte Paare jedes Mal neu zu übertragen, werden in den kodierten Headern nur indexierte Nummern hinzugefügt (Peon/Ruellan 2015, 4), (Grigorik 2015, 19-20).

Statische und Dynamische Tabellen existieren sowohl auf der Client- als auch auf der Serverseite und werden innerhalb der bestehenden Verbindung laufend aktualisiert. Jedes neue Schlüssel - Werte Paar wird entweder zur existierenden Tabelle hinzugefügt oder zuvor übertragene Werte in der Tabelle werden ersetzt (Buch IG, 222). Außer statischen und dynamischen Tabellen werden die Headerfelder während der Übertragung mithilfe des statischen Huffman – Code kodiert, um die Gesamtgröße zu reduzieren (Grigorik 2015, 19 - 20).

Wie zuvor erwähnt wurde, wird die Definition von Requests- und Responses- Headerfeldern im HTTP/2 – Protokoll nicht verändert. Nur zwei Änderungen im HTTP/2 – Protokoll haben die Darstellung der Headerfelder betroffen. Erstens, dass alle Headerfelder klein geschrieben werden. Zweitens werden die Requests – Headerfelder „:authority", „:method", „:path" und „:scheme" als pseudo - Headerfelder dargestellt werden (Grigorik 2015, 20).

3.3.5. Eine TCP – Verbindung pro Domain

Um im HTTP/1.1 – Protokoll die Request und Response Parallelisierung zu ermöglichen, werden mehrere TCP – Verbindungen (je nach Browser) gleichzeitig geöffnet. Im Gegenteil werden im

HTTP/2 – Protokoll alle „Streams“ innerhalb einer TCP – Verbindung multiplexiert. Deshalb ist es nicht mehr nötig, mehrere Verbindungen gleichzeitig zu halten (Grigorik 2015, 13).

Die Wiederverwendung einer TCP – Verbindung hat mehrere Vorteile:

- Innerhalb einer TCP – Verbindung können die Streams besser priorisiert werden. Außerdem wird die Zeit für jede einzelne Verbindungseinrichtung (Three - Way - Handshake) gespart. Wenn die Verbindung durch eine SSL – Verschlüsselung läuft, wird diese Zeitersparnis noch deutlicher sein.
- Bessere Kompressionsmöglichkeiten („Header Kompression“) innerhalb einer TCP – Verbindung.
- Je weniger Verbindungen gleichzeitig bearbeitet werden, desto weniger CPU- und RAM-Leistungen werden verbraucht. Dies betrifft sowohl den Client als auch den Server.

(Grigorik 2015, 13 - 14).

Andererseits wird die Verwendung einer TCP – Verbindung nicht nur vorteilhaft sein:

- Falls das TCP – Paket verloren geht, wird die Kapazität der TCP – Fenstergröße (TCP Receive Window Size) automatisch verringert (siehe Unterkapitel „HTTP/1 – Optimierungstechniken, Stand bisher“). Dies führt dazu, dass die maximale Datenrate der kompletten TCP – Verbindung verringert wird und die Daten den Client verzögert erreichen werden.
- Das „Head-Of-Line Blocking“ – Problem wurde zwar von der HTTP – Schicht genommen (durch die Multiplexierung), findet aber immer noch in der TCP – Schicht statt. Dies passiert, da beim Verlust eines TCP – Pakets auf dem Weg zum Empfänger, die gesamte Sequenz von TCP – Paketen im TCP – Puffer gehalten wird, bis das verlorene TCP – Paket den Empfänger erreichen wird (Buch IG, 216). Weil innerhalb des HTTP/2 – Protokolls nur eine TCP – Verbindung zur Verfügung steht, können diese Verzögerungen deutlich ausfallen.
- Falls die TCP – Fensterskalierung (TCP Window Scale Option) deaktiviert ist, wird die gesamte Datenrate der TCP – Verbindung durch Verzögerungen in der Bandbreite limitiert.

(Grigorik 2015, 14).

Aus den zuvor beschriebenen Vor- und Nachteilen können mögliche Szenarien gedacht werden, in denen wenige TCP – Verbindungen vorteilhafter sein können. Es gab Tests, in denen bestätigt wurde, dass die negative Wirkung des „Head-Of-Line Blocking“ die Vorteile der „Header Kompression“ und der Priorisierung überwiegt, wenn TCP – Pakete während der Verbindung verloren gehen (Grigorik 2015, 14 - 15).

3.3.6. Flow Control

Die Möglichkeit, um innerhalb des neuen Protokolls Ressourcen zu multiplexieren, verursacht einen Wettbewerb zwischen allen Ressourcen, die die Bandbreite zwischen sich aufteilen müssen. Prioritäten, die für „Streams" gesetzt werden können, sind dafür nicht effizient genug, um die Verteilung zwischen den Streams zu kontrollieren. Um den „Stream" – Durchfluss zu verteilen und damit das Problem zu lösen, gibt es innerhalb des HTTP/2 – Protokolls einen einfachen Mechanismus: „Flow Control". „Flow Control" basiert auf den WINDOW_UPDATE – Frames. Der Empfänger kündigt an, wie viele Bytes innerhalb des „Streams" und der Sitzung er annehmen kann. Die Fenstergröße des „Flow Controls" wird während der Sitzung verändert und WINDOW_UPDATE – Frames übertragen die „Stream ID" und die Fenstergröße für die einzelnen „Streams". Außerdem kann der Empfänger jede Fenstergröße für einzelne „Streams" innerhalb der bestehenden Verbindung auswählen. „Flow Control" kann beim Empfänger sowohl für einzelne „Streams" als auch für eine komplette Sitzung deaktiviert werden (Buch IG, 219).

Soweit die Verbindung unter dem HTTP/2 – Protokoll aufgebaut wird, werden Client und Server den SETTINGS – Frame miteinander austauschen. In diesen „Frames" wird die Fenstergröße des „Flow Controls" für beide Seiten der Kommunikation gesetzt. Innerhalb der Spezifikation des HTTP/2 – Protokolls gibt es keine bestimmten Algorithmen und keine definierten Parameter für die WINDOW_UPDATE – Frames. Es ist dem Entwickler überlassen, die Größe der anzunehmenden Daten einzustellen (Buch IG, 219).

Der Mechanismus des „Flow Controls" ist zum Mechanismus der „Flow Control" Funktion des TCP – Protokolls sehr ähnlich. „Flow Control" kann aber nicht einzelne „Streams" innerhalb einer TCP – Verbindung unter Kontrolle nehmen. Aus diesem Grund gibt es den „Flow Control" für das HTTP/2 – Protokoll (Buch IG, 219).

3.3.7. Server Push

„Server Push" ist eine mächtige Funktion des HTTP/2 – Protokolls. „Server Push" ist die Fähigkeit des Servers, zu den vom Client angefragten Ressourcen noch weitere Ressourcen zu liefern. D.h., dass mithilfe von „Server Push" der Client mit Erhalt der angefragten Ressource noch weitere Ressourcen zurückbekommt. Dadurch müssen für die anderen Ressourcen keine weiteren Anfragen gestellt werden. Abb. 16 verdeutlicht die Funktionsweise des „Server Pushs" (Grigorik 2015, 17).

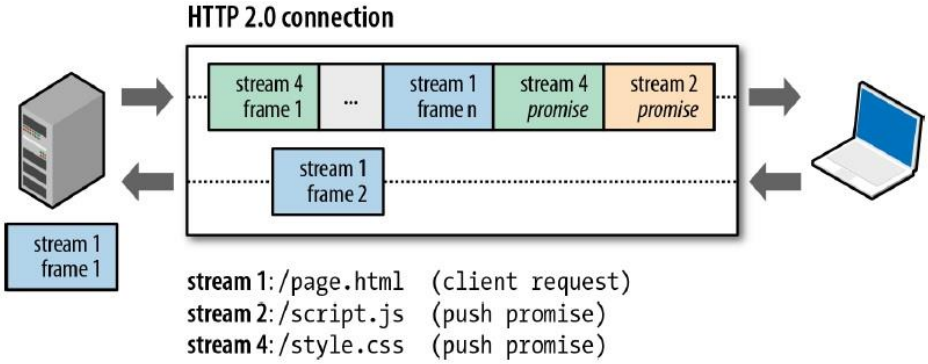

Abb. 16: Funktionsweise des „Server Pushs" (Buch IG, 220).

Warum könnte diese Funktion so nützlich sein? Eine typische Webapplikation besteht aus dutzenden Ressourcen, die nacheinander entdeckt und angefragt werden. „Server Push" spart die Anfragezeit an den Server (Grigorik 2015, 17). Besonders nützlich kann „Server Push" dann sein, wenn dem Server bekannt ist, dass der Client für bestimmte Ressourcen eines bestimmten Requests unmittelbar darauf weitere Ressourcen braucht und bald selbst nach ihnen fragen wird (Belshe et al. 2015, 60-61).

Alle Ressourcen, die gepusht werden sollen, werden vom Server in Abhängigkeit von einem vom Client zuvor initiierten Requests geschickt (Belshe et al. 2015, 60). Außerdem müssen sie die gleiche Herkunft (gleiche Domain) wie der Request haben (Buch IG, 221).

Soweit die HTTP/2 – Verbindung aufgebaut wird, werden der Client und der Server die SETTINGS – Frames austauschen, in denen steht, wie viele „Streams" in beiden Richtungen gleichzeitig geliefert werden können. Deswegen kann der Client die Anzahl an gepushten „Streams" begrenzen oder überhaupt abschaffen, wenn diese Anzahl auf null gestellt wird (Buch IG, 220).

Um im HTTP/1.1 – Protokoll die Anzahl von Requests und die Anfragezeit zu sparen, wurden die Dateien per Ressource Inlining in der HTML – Datei hinzugefügt. Im Gegenteil zu den Optimierungstechniken für das HTTP/1.1 – Protokoll (siehe Unterkapitel „3.1. HTTP/1 – Optimierungstechniken, Stand bisher") hat „Server Push" unter dem HTTP/2 – Protokoll mehrere Vorteile:

- Gepushte Ressourcen können beim Client in den Cache gelegt werden.
- Gepushte Ressourcen können zwischen Unterseiten einer Webapplikation benutzt werden.
- Gepushte Ressourcen können mit anderen Ressourcen multiplexiert werden.
- Gepushte Ressourcen können vom Server priorisiert werden.

(Grigorik 2015, 18).

„Streams", die durch „Server Push" geöffnet werden, werden durch PUSH_PROMISE – Frames initiiert. Dieses „Frame" signalisiert die Bereitschaft des Servers, um noch andere in den Einstellungen beschriebene Ressourcen zusätzlich zum Response (DATA – Frame der angefragten

Ressource) zu liefern. Der PUSH_PROMISE – Frame besteht vor allem aus typischen „Frame Headern" und einer zusätzlichen „Promised - Stream ID", die diesen Stream eindeutig identifiziert. Alle PUSH_PROMISE – Frames erreichen den Client vor dem DATA – Frame des eigentlichen Requests. Diese Reihenfolge ist wichtig, weil dies dem Client das Signal gibt, nicht mehr selber nach diesen Dateien zu fragen. Nachdem der Client diesen „Frame" bekommt, hat er die Option, den ganzen „Stream" zu verwerfen, falls z.B. die Datei schon im Browser Cache liegt. Dies passiert mithilfe des RST_STREAM – Frames (Grigorik 2015, 18).

Der Client hat auch die Möglichkeit, die Anzahl von gepushten Ressourcen zu begrenzen. Diese Einstellungen wurden innerhalb des SETTINGS – Frames übertragen, der nach dem Aufbau der HTTP/2 – Verbindung ausgetauscht wird und der sich jeder Zeit innerhalb der Sitzung ändern kann (Grigorik 2015, 18).

3.3.8. Verbindungsaufbau unter dem HTTP/2 – Protokoll

Wie zuvor erwähnt wurde, müssen, um das neue Protokoll benutzen zu können, beide Seiten der Kommunikation binäre Kodierung unterstützen (Grigorik 2015, 6). Da nicht alle Browser und Browserversionen das neue Protokoll unterstützen (Abb. 9), muss zwischen dem Client und dem Server eine „Verabredung" über die Nutzung des geeigneten Protokolls stattfinden. Sollte eine von beiden Seiten der Kommunikation das neue Protokoll nicht unterstützen, wird das HTTP/1.1 – Protokoll benutzt. Soweit der Client das HTTP/2 – Protokoll unterstützt, muss er herausfinden, ob der Server das gleiche tut.

Unter dem HTTP/2 – Protokoll ist es auch möglich, die Webapplikation unter unverschlüsselter Verbindung aufzurufen, da dies im HTTP/2 – Standard festgehalten ist (Eissing 2015). Wenn HTTP/1.1- und HTTP/2 – Protokolle unter dem gleichen Port (80) laufen und wenn es keine anderen Informationen darüber gibt, ob der Server das HTTP/2 – Protokoll unterstützt, wird der Client einen „Upgrade" – Mechanismus zur Nutzung des HTTP/2 – Protokolls durchführen. Dadurch wird eine „Vereinbarung" für die weitere Kommunikation getroffen. Dieser Mechanismus hat den folgenden Ablauf: (Grigorik 2015, 21-22)

Clientseitiger Request:

GET /page HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP/2-Settings
Upgrade: h2c **(1)**
HTTP/2-Settings: (SETTINGS payload) **(2)**

Serverseitiger Response:

HTTP/1.1 200 OK **(3)**
Content-length: 243
Content-type: text/html
(... HTTP/1.1.1 response ...)

(oder)

HTTP/1.1 101 Switching Protocols **(4)**
Connection: Upgrade
Upgrade: h2c
(... HTTP/2 response ...)

(1): Es wird ein HTTP/1.1 Request mit HTTP/2 „Upgrade" – Header initialisiert, um zu zeigen, dass der Client das HTTP/2 – Protokoll unterstützt (Grigorik 2015, 21-22).

(2): Der Client schickt gleich einen binären SETTINGS – Frame mit den HTTP/2 – Einstellungen (Grigorik 2015, 21-22).

(3): Wenn der Server das HTTP/2 – Protokoll nicht unterstützt, wird der Response durch das HTTP/1.1 – Protokoll zurückgeliefert (Grigorik 2015, 21-22).

(4): Im Fall, dass der Server das HTTP/2 – Protokoll unterstützt, wird er den „101 Switching Protocols" – Response zurückgeben und sofort zur unverschlüsselten HTTP/2 – Verbindung (h2c) wechseln (Grigorik 2015, 21-22).

Wenn der Client durch vorherige Aufrufe der Webapplikation gemerkt hat, dass entsprechender Server das HTTP/2 – Protokoll unterstützt, wird der „Upgrade" – Mechanismus nicht wieder stattfinden, sondern die Kommunikation wird gleich unter dem HTTP/2 – Protokoll anfangen (Grigorik 2015, 21-22).

Oben wurde der Fall beschrieben, wie die Kommunikation zwischen Client und Server unter unverschlüsselten Verbindung stattfinden kann. Im Anhang befindet sich unter „Upgrade HTTP/2" der Code von Requests und Responses der Webapplikation, welche für die Tests in dieser Masterarbeit benutzt wird. Diese wurde am lokalen PC mithilfe des „Curl" – Clients aufgerufen.

Allerdings muss man beachten, dass die Browser, die das HTTP/2 – Protokoll unterstützen, nur verschlüsselte (TLS) Verbindungen erlauben (Grigorik 2015, 21), (NGINX, Inc. 2015, 3). Aus diesem Grund wird der oben beschriebene Fall im Browser nicht funktionieren und die Kommunikation wird nur unter dem HTTP/1.1 – Protokoll laufen.

Um die verschlüsselte Kommunikation zwischen Browser und Server unter dem HTTP/2 – Protokoll erlauben zu können, benötigt man die TLS – Verschlüsselung mit der Erweiterung „Application-Layer Protocol Negotiation" (kurz: ALPN) (Buch IG, 226). ALPN ist notwendig, um das Protokoll zwischen dem Client und dem Server auszuhandeln, das über eine verschlüsselte Verbindung laufen soll. Hierbei werden zusätzliche „Round Trips" und als Folge dessen unnötige Latenz zwischen Client und Server vermieden (Wikipedia 2016g), (Buch IG, 53-54).

Um die Kommunikation unter einer TLS – verschlüsselten Verbindung mit dem HTTP/2 – Protokoll zu zeigen, wird der Code von Requests und Responses der aufgerufenen Test-Webapplikation gespeichert, die für diese Masterarbeit entwickelt wurde. Dieser befindet sich im Anhang unter „9.1 Upgrade HTTP/2". Die Webapplikation wurde am lokalen PC mithilfe des „Curl" – Clients aufgerufen. Im ersten Fall, die Webapplikation wird mit TLS – Verschlüsselung aufgerufen, ist zu sehen, dass die TLS – Verbindung mit „ALPN" benutzt wird und „ALPN" die Verwendung des neuen Protokolls erlaubt. Dadurch wird die Kommunikation unter dem HTTP/2 – Protokoll laufen. Wenn die Webapplikation ohne Verschlüsselung aufgerufen wird, wird die weitere Kommunikation unter dem HTTP/1.1 – Protokoll laufen. Man sieht, dass der Client beim Request kein „Upgrade" – Header initialisiert. Deshalb hat der Server das HTTP/1.1 – Protokoll nicht auf das HTTP/2 – Protokoll gewechselt.

3.4. Mögliche Optimierungstechniken für das HTTP/2 – Protokoll

Im vorherigen Unterkapitel wurde die Funktionsweise des HTTP/2 – Protokolls beschrieben. In diesem Teil der Masterarbeit wird auf mögliche Frontend- und Backend-Optimierungstechniken für das neue Protokoll aufmerksam gemacht. Man muss die Funktionsweise des neuen Protokolls verstehen, um einschätzen zu können, welche Optimierungstechniken geeignet sein können. Deshalb wird ein kurzer Überblick über neue Funktionen des HTTP/2 – Protokolls aufgelistet:

- Innerhalb des HTTP/2 – Protokolls wird nur eine TCP – Verbindung benötigt.
- Alle HTTP – Header werden während dem Transport komprimiert.
- Alle Requests und Responses werden auf kleine „Frames" aufgeteilt und innerhalb der Sitzung multiplexiert.
- Es gibt die Möglichkeit, um die Ressourcen per „Server Push" zu übergeben.
- Alle „Streams" können priorisiert werden.

(Grigorik 2015, 6-23).

Ausgehend von der Funktionsweise des neuen Protokolls, kann man überlegen, was man im Vergleich zu den Frontend – Optimierungstechniken des HTTP/1.1 – Protokolls unternehmen kann. Da es im HTTP/2 – Protokoll nur eine TCP – Verbindung gibt, kann es sein, dass innerhalb dieser Verbindung alle Ressourcen besser priorisiert werden können, als bei mehreren Verbindungen (Grigorik 2013, 242). Wenn die Ressourcen nicht mehr auf unterschiedliche Domains aufgeteilt werden, wird der Client genauer wissen, welche Ressourcen er zuerst bekommen

möchte und kann die Ressourcen nach Prioritäten sortieren. In der Theorie können sowohl clientseitige als auch serverseitige „Streams" priorisiert werden (Grigorik 2015, 18), (Buch IG, 216). In der Dokumentation des „mod_http2"-Moduls des Apache-Servers steht, dass „Streams", die clientseitig geöffnet werden, vom Client priorisiert werden müssen. Nur im Fall des „Server Pushes" werden serverseitig geöffnete „Streams" vom Server priorisiert (Apache Software Foundation 2016a).

Wenn man die Funktionsweise der dynamischen Tabellen in „HPACK" genauer betrachtet, wird man sehen, dass innerhalb einer TCP – Verbindung diese effizienter benutzt werden können (Grigorik 2013, 242). Der Grund dafür ist, dass dynamische Tabellen für eine Verbindung weniger Inhalte haben werden.

Aufgrund der Funktionsweise der Priorisierungen des „Streams" und „HPACK" kann festgestellt werden, dass die Aufteilung der Ressourcen einer Webapplikation auf unterschiedliche Domains im Fall der Verwendung des HTTP/2 – Protokolls nur schaden wird (Grigorik 2013, 242). Aus diesem Grund wird diese Optimierungstechnik für die Test-Webapplikation nicht angewendet werden.

Innerhalb des HTTP/1 – Protokolls ist es immer empfehlenswert, mehrere Dateien gleichen Typs in einer größeren Datei zusammenzufassen, damit Requests gespart werden (Grigorik 2013, 241). Es stellt sich die Frage, ob diese Maßnahme auch für das HTTP/2 – Protokoll angewendet werden muss. Theoretisch gesehen, sollte es für das HTTP/2 – Protokoll nicht wichtig sein, ob die Ressourcen zusammengefasst sind oder nicht, weil alle Ressourcen auf einzelne „Frames" aufgeteilt und miteinander multiplexiert werden. Die Zusammenfassung von Dateien (inkl. Sprites für Bilder) hat viele Nachteile, die im Unterkapitel „3.1. HTTP/1.1 – Optimierungstechniken, Stand bisher" aufgelistet sind. Aus diesem Grund ist es empfehlenswert, bei der Verwendung des HTTP/2 – Protokolls Dateien einzeln zu halten (Grigorik 2013, 243).

Noch eine gewöhnliche Optimierung für das HTTP/1 – Protokoll ist das „Ressourcen Inlining" in der HTML Datei, wenn diese für die Erstdarstellung kritisch sind. Dadurch werden zusätzliche Requests gespart und Inhalte schnellstmöglich an den Client übergeben. Dies hat ähnliche Nachteile, wie die Zusammenfassung mehrerer Dateien (NGINX, Inc. 2015, 13). Da es innerhalb des HTTP/2 – Protokolls die „Server Push" Funktion gibt, wird diese Optimierung unnötig und eher schädlich sein. Mithilfe des „Server Pushs" können einzelne Dateien unabhängig bearbeitet, gecached und für jede Seite der Webapplikation benutzt werden. Außerdem kann die Verwendung des „Server Pushs" sehr vorteilhaft unter mobilen Netzwerkverbindungen sein, weil dadurch viele Anfragen gespart werden (Grigorik 2013, 243).

Serverseitige Optimierungstechniken

Der Fokus des HTTP/2 – Protokolls liegt auf der Verbesserung des Datentransports, der Ermöglichung besseren Datendurchsatzes und der niedrigeren Latenz zwischen Client und Server. Außer den zuvor beschriebenen möglichen Optimierungstechniken ist es auch wichtig, möglichst gute Bedingungen für die Funktionsweise von TCP und TLS einzurichten. Aus diesem Grund muss wenigstens auf drei serverseitige Einstellungen geachtet werden. Zuerst muss der Server für die TCP – Verbindung die „cwnd" – Startgröße von mind. 10 „TCP – Segmenten" unterstützen (zur Begründung siehe Kapitel „3.1 HTTP/1.1 – Protokoll: Optimierungstechniken, Stand bisher"). Außerdem muss der Server die TLS – Verschlüsselung mit Erweiterung von ALPN unterstützen, um zusätzliche „Round-Trip Times" zu vermeiden. Sodann muss der Server Mechanismen unterstützen, die beim „TLS – Handshake " die Latenz reduzieren (Grigorik 2013, 241-242).

In der aktuellen Arbeit liegt der Fokus auf Frontend – Optimierungstechniken. Deshalb werden dieselben serverseitigen Einstellungen (bezüglich oben beschriebenen serverseitigen Optimierungstechniken) sowohl für die Anwendung des HTTP/1.1- als auch für das HTTP/2 – Protokoll für die Test-Webapplikation angewendet. Für Untersuchungen unter verschlüsselter Verbindung wurde die TLS – Verschlüsselung mit Erweiterung von ALPN verwendet. Die „cwnd" – Startgröße beträgt 16 „TCP – Segmente".

Serverseitige Anforderungen

Um am Ende eine perfomante Webanwendungen haben zu können, muss man nicht nur geeignete Optimierungstechniken anwenden, sondern noch ganz genau aufpassen, wie gut der Server und der Client oben beschriebene Funktionen bearbeiten. Es ist schwierig, um die Funktionen des Clients zu beobachten und zu beeinflussen. Die Funktionsweise jedes Clients unterscheidet sich je nach Entwicklung und Version.

Der Webserver spielt bei der Ausarbeitung des HTTP/2 – Protokolls jedoch eine große Rolle. Wie qualitativ die Funktionen des HTTP/2 – Protokolls verarbeitet werden, ob „Server Push" vorhanden ist, und wie gut die Multiplexierung der einzelnen „Frames" funktionieren wird, liegt an der Qualität der Implementierung des Webservers (Grigorik 2013, 247). Aus diesem Grund muss der Webserver genau ausgewählt werden:

- Der HTTP/2 – Webserver muss die vom Client gesetzten Prioritäten der „Streams" verstehen, verarbeiten und in entsprechender Reihenfolge zurückliefern.
- Der HTTP/2 – Webserver muss „Server Push" unterstützen.
- Der HTTP/2 – Server muss mehrere Strategien der Implementierung des „Server Pushs" unterstützen.

(Grigorik 2013, 247).

Wie der Webserver ausgewählt wird, wird im Kapitel „4. Vorbereitung der Tests und zur Testevaluation“ beschrieben.

Zusammenfassung: Mögliche Optimierungstechniken für das HTTP/2 – Protokoll

Oben wurden mögliche sowohl client- als auch serverseitige Optimierungstechniken unter dem HTTP/2 – Protokoll beschrieben. Es werden unter anderem einige Empfehlungen für clientseitige Techniken zur Performance – Optimierung vorgestellt, die unter dem neuen Protokoll gemacht werden sollten. Im praktischen Teil dieser Masterarbeit wird darauf geachtet, ob es sich bestätigt, dass einzeln gehaltene (nicht wie unter dem HTTP/1.1 - Protokoll zusammengefasste) Ressourcen für die Performanz der Webapplikation unter dem neuen Protokoll besser sind.

Außerdem wird im praktischen Teil dieser Masterarbeit darauf geachtet, wie gut der Webserver oben beschriebene Funktionen, wie Multiplexierung von „Frames“, „Server Push“ und Priorisierung von gepuschten Ressourcen verarbeitet. Bei per „Server Push“ übergebenen Ressourcen ist es auch wichtig, dass diese nach dem ersten Aufruf der Webapplikation im Browser-Cache landen werden. Im praktischen Teil der Arbeit wird verglichen, wie die in diesem Kapitel beschriebenen theoretischen Aussagen zur aktuellen Implementierung des neuen Protokolls passen.

4. Vorbereitung der Tests und zur Testevaluation

4.1. Vorbereitung der Tests

Im aktuellen Unterkapitel wird beschrieben, wie die Tests aufgebaut und vorbereitet werden und wie der Server eingestellt wird. Deshalb werden in diesem Teil der Masterarbeit folgende Fragen bearbeitet:

- Welche Webapplikation ist für die Untersuchung der Fragestellung der aktuellen Masterarbeit am besten geeignet?
- Wie soll die Webapplikation für die zwei Protokolle HTTP/1.1 und HTTP/2 optimiert werden, damit man für jedes Protokoll bestmögliche Ergebnisse erreichen kann?
- Welcher Server und welche Serverversion eignen sich für die Tests? Was sind die wichtigsten Voraussetzungen für den ausgewählten Server?
- Welche Maßnahmen müssen für die Verwendung des HTTP/2 – Protokolls unternommen werden? Wie soll der Server konfiguriert werden?

Welche Webapplikation ist für die Untersuchung der Fragestellung der aktuellen Masterarbeit am besten geeignet?

In der aktuellen Masterarbeit liegt der Fokus auf Frontend-Optimierungstechniken im Bereich des kritischen Rendering – Pfades. Welche Webapplikation kann am besten zum Testen genommen werden, um die Wirkung dieser Techniken besser beobachten zu können? Aus der User Experience-Sicht ist es wichtig, dass nicht nur die Webapplikation selbst schnellstmöglich geladen wird, sondern auch die Reihenfolge, in der die einzelnen Komponenten dieser Webapplikation geladen werden, ist wichtig. Besonders wichtig sind die Komponenten, die sich innerhalb des Viewports (aktueller Viewport) befinden (Abb. 17) (Sexton 2015).

Abb. 17: Primäre und sekundäre Komponenten einer Webseite innerhalb des Viewports (<https://varvy.com/pagespeed/critical-render-path.html>)

Der Browser kann unterschiedliche Prioritäten, je nach Typ der Ressource, setzen (Grigorik 2013, 172). Aus diesem Grund werden verschiedene Seitenkomponenten je nach Ressourcentyp in einer unterschiedlichen Reihenfolge auf dem Viewport erscheinen. Deshalb wurde entschieden, eine Webapplikation mit vielen unterschiedlichen Ressourcentypen zur Untersuchung zu nehmen. Dadurch kann besser beobachtet werden, in welcher Reihenfolge die

Webapplikation geladen wird. Das Ziel ist natürlich, dass der erste Eindruck des Nutzers zur geladenen Webapplikation positiv bleiben wird.

Es muss also eine Webapplikation entwickelt werden, die aus möglichst vielen Ressourcentypen besteht, die auf dem Viewport des Browsers zusammen passen werden: CSS Dateien, JavaScript Dateien, Bilder und Schriften. Vor 1,5 Jahren wurde eine Internetseite für die Masterstudiengänge „Audiovisuelle Medien", „Medienwirtschaft" und „Unternehmenskommunikation" der Hochschule der Medien von mehreren Studenten (inkl. der Verfasserin dieser Masterarbeit) entwickelt (Medienmaster.de 2016). Die Startseite verfügt über oben beschriebene Kriterien. Diese Webapplikation wurde unter dem Content-Management-System „Wordpress" entwickelt. Im Rahmen dieser Masterarbeit wurde entschieden, statische Ressourcen zu nehmen, weil die Bearbeitungszeit dynamischer Sprachen (PHP) und ständige Aufrufe der Datenbank auf die in den Tests zu untersuchenden Parameter einen großen Einfluss haben kann. Aus diesem Grund wird nur die Struktur dieser Webapplikation übernommen und in vielen Stellen zusätzlich angepasst.

So wird eine statische Webapplikation zum Testen erstellt. Diese ist unter: http://im-im2.hdm-stuttgart.de/ oder 141.62.110.42 erreichbar. Deren Startseite besteht aus mehreren Komponenten (von oben nach unten): ein Logo, ein Menü, eine großgeschriebene Zeile mit zwei unterschiedlichen Schriftarten, ein JavaScript-Slider mit zwei großen Bildern und zwei verkleinerten Bildern (Thumbnails), zwei Reihen von Studienganglogos (Bilder mit daruntergeschriebenem Text) und ein Footer von Jahrgangssponsoren. Die Startseite beinhaltet insgesamt fünf Schriftarten.

Im Viewport eines Browsers am lokalen PC sind alle Seitenkomponenten bis zum Footer zu sehen. Dies bedeutet, dass bei der Untersuchung der Ladezeit der Webapplikation beobachtet werden kann, wie schnell und in welcher Reihenfolge unterschiedliche Ressourcentypen der Startseite geladen werden.

Außer der Startseite wurden noch zwei Unterseiten zum Testen hinzugefügt. Diese sind „dozenten.html" und „dozenten2.html". Sie beinhalten das Logo, ein Menü, Navigationszeile und Bilder von Dozenten aus oben beschriebenen Masterstudiengängen. Wenn man mit der Maus über einzelne Bilder fährt, sieht man detaillierte Informationen zum einzelnen Dozenten. Die Unterseiten „dozenten.html" und „dozenten2.html" haben nur einen einzigen Unterschied: in „dozenten.html" werden alle Bilder der Dozenten durch zusätzliche CSS Dateien geladen und in „dozenten2.html" werden alle Bilder der Dozenten in der HTML Datei hinzugefügt. Dies wurde gemacht, um zu schauen, ob die Ressourcenreihenfolge sich zwischen beiden Unterseiten unterscheiden wird. Diese Seiten sind auch deshalb interessant für die Untersuchung, weil diese bildlastig sind.

Wie soll die Webapplikation für die zwei Protokolle HTTP/1.1 und HTTP/2 optimiert werden, damit man für jedes Protokoll bestmögliche Ergebnisse erreichen kann?

Wie im theoretischen Teil beschrieben wurde, wurden oftmals für die Fronend-Optimierung unter dem HTTP/1.1 – Protokoll mehrere Dateien eines Datentypes in einer Datei zusammengefasst, um die Anzahl von Requests zu reduzieren. Da diese Zusammenfassung viele Nachteile mit sich bringt, wurde für das HTTP/2 – Protokoll empfohlen, die Dateien einzeln zu halten.

Aus diesem Grund werden alle Ressourcen sowohl auf der Startseite als auch bei den Unterseiten für jedes Protokoll unterschiedlich aufgeteilt. Von den Inhalten wird die Webapplikation gleich sein.

Es gibt zwei CSS Dateien, die sowohl unter Verwendung des HTTP/1.1- als auch des HTTP/2 – Protokolls unverändert benutzt werden: „style.css" und „slider.css". Unter Verwendung des HTTP/1.1 – Protokolls entstehen noch drei CSS Dateien: „animate.css", „bootstrap.css" und „stylehttp1.css". Diese drei Dateien wurden bei Verwendung des HTTP/2 – Protokolls zusätzlich aufgeteilt: „animate.css" wurde auf „animate1.css" und „animate2.css"aufgeteilt; „bootstrap.css" wurde auf „bootstrap1.css", „bootstrap2.css" und „bootstrap3.css" aufgeteilt; „stylehttp1.css" wurde auf „stylehttp2.css", „menu.css", „footer.css" und „team.css" oder „team2.css" für „dozenten.html" und „dozenten2.html" aufgeteilt. So entstehen bei Verwendung des HTTP/1.1 – Protokolls fünf CSS Dateien und bei Verwendung des HTTP/2 – Protokolls insg. 12 CSS Dateien.

Die „bootstrap.min.js" JavaScript Datei wird auf einzelne Bootstrap-Komponenten aufgeteilt und zusätzlich minimiert, sodass unter Verwendung des HTTP/1.1 – Protokolls eine JavaScript Datei benutzt wird und unter Verwendung des HTTP/2 – Protokolls 11 Dateien benutzt werden. Diese Maßnahme ist für die eigentliche Entwicklung nicht nötig, hier wird damit nur ein Fall mit vielen JavaScript Dateien simuliert. Für beide Protokolle werden noch vier JavaScript Dateien für die Webapplikation benutzt: „jquery.min.js" und auf der Startseite noch: „jssor.slider.mini.js", „slider.js" und „wow.min.js".

Für die Unterseiten „dozenten.html" und „dozenten2.html" werden zusätzlich die Bilder unterschiedlich aufgeteilt. Unter Verwendung des HTTP/1.1 – Protokolls werden alle kleinen Bilder in einer Datei (Sprite) zusammengefasst, währenddessen unter Verwendung des HTTP/2 – Protokolls alle Bilder als einzelne kleine Bilder gehalten werden.

Welcher Server und welche Serverversion eignen sich für die Tests? Was sind die wichtigsten Voraussetzungen für den ausgewählten Server?

Es gibt mehrere Webserver, die das HTTP/2 – Protokoll implementieren (Thomson/Nottingham 2016). In den ersten Schritten dieser Masterarbeit wird überlegt, welcher Webserver sich am besten anwenden lässt. Dafür wird geschaut, welche Webserver am häufigsten verwendet werden. Abb. 18 veranschaulicht den prozentualen Anteil der sechs populärsten Webbrowser. Es

kann von Vorteil sein, um den Server zu nehmen, an dem schon Tests und Untersuchungen zur Verwendung des HTTP/2 – Protokolls durchgeführt wurden.

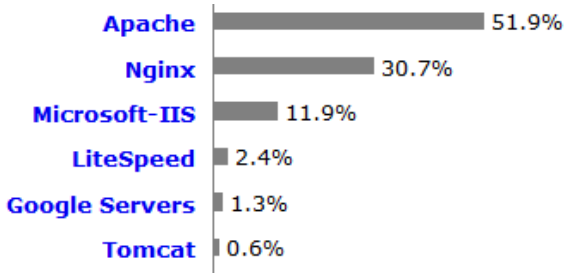

Abb. 18: Prozentualer Anteil der meist verwendeten Webserver, Stand: 17.08.2016 (<https://w3techs.com/technologies/overview/web_server/all>)

Bei der Suche nach existierenden Beispielen und Implementierungen des HTTP/2 – Protokolls werden oftmals Testbeispiele unter dem „H2O"-Server (DeNA Co., Ltd. 2015) oder dem „NGINX"-Server (NGINX, 2016) gefunden.

Im Fokus dieser Masterarbeit liegt u.a. die Untersuchung der „Server Push" – Funktion des HTTP/2 – Protokolls. Deshalb ist die Anwesenheit dieser Funktion ein wichtiges Kriterium zur Serverwahl. In der elektronischen Ausgabe „HTTP/2 for Web Application Developers" des „NGINX"-Servers heißt es am 16.09.2015, dass die aktuelle Serverimplementierung (NGINX Version 1.9.5) den „Server Push" nicht unterstützt (NGINX, 2015). Nach fast einem Jahr hat sich dieses Status nicht verändert. Aus diesem Grund kann der „NGINX"-Server im Rahmen dieser Masterarbeit nicht genommen werden.

Obwohl ein paar Tests und Untersuchungen des „H2O"-Webservers gefunden wurden, wurde auf Grund der mangelnden Erfahrung mit diesem Server und einer nicht ausführlichen Dokumentation dieses Servers ein anderer Server genommen.

Aus Abb. 18 ist zu sehen, dass der Apache-Server der meist verwendete Webserver ist, der jedes zweite Mal verwendet wird. Außerdem gab es schon ein wenig Erfahrung mit diesem Server. Zu der Zeit, als der Server ausgewählt wurde, wurden keine öffentlichen Tests, Beispiele oder Untersuchungen des HTTP/2 – Protokolls unter dem Apache-Server gefunden. Für die HTTP/2 – Implementierung ist das Apache-Modul „mod_http2" verantwortlich. Dieses stellt unter anderem „Server Push" zur Verfügung. Aus diesen Gründen wurde entschieden, den Apache-Webserver als Webserver für die Untersuchungen zu nehmen.

Welche Maßnahmen müssen für die Verwendung des HTTP/2 – Protokolls unternommen werden? Wie soll der Server konfiguriert werden?

Das Apache-Modul „mod_http2" ist nur ab der Apache-Version 2.4.17 verfügbar. Allerdings ist zu beachten, dass sich zum Zeitpunkt dieser Masterarbeit das Modul im Status „experimentell" befand. Dies bedeutet, dass sich sein Verhalten, seine Direktiven und Standardeinstellungen von Version zu Version noch stark ändern können (Apache Software Foundation 2016a).

Welche Version des Apache-Servers wird genommen? Die Implementierung des HTTP/2 – Protokolls ist ab der Apache-Version 2.4.17 verfügbar. Es ist nun empfehlenswert, um neuere Versionen zu nehmen, weil einige Funktionen und Direktiven verbessert und ergänzt wurden (Apache Software Foundation 2016b). Die Serverinstallation und Serverkonfiguration für die Tests wurde im Mai 2016 vollzogen. Damals war die Apache-Version 2.4.21 gerade erschienen. Aus Gründen möglicher Instabilität der neuesten Version wurde die nächstältere Version des Apache-Servers 2.4.20 bevorzugt.

Um Apache 2.4.20 installieren zu können, benötigt man ein entsprechendes Betriebssystem, das genau diese Serverversion unterstützt. Wenn diese Version manuell installiert und konfiguriert wird, würden keine automatischen Updates durchgeführt. Falls Sicherheits- oder Funktionslücken in der ausgewählten Version auftreten würden, müsste man diese auch manuell lösen können.

Deshalb bestand der nächste Schritt darin, das passende Betriebssystem für die möglichst neueste Apache-Version zu finden. Es standen nur „Linux"- Betriebssysteme zur Diskussion. Das Betriebssystem wurde mithilfe der Liste der meist verwendeten „Linux"-Betriebssysteme (DistroWatch 2016) ausgewählt. Es gab zwei wichtige Auswahlkriterien zum Betriebssystem: große Community und detaillierte Endanwenderdokumentation. Nach der Recherche zur Unterstützung neuerer Apache-Versionen wurde festgestellt, dass nur wenige „Linux"-Betriebssysteme oben beschriebenen Kriterien entsprechen. Es wurden nur zwei solche Betriebssysteme gefunden: „Fedora" (Red Hat, Inc. and others 2015) und später „openSUSE" (SUSE LLC 2015).

Die erste Wahl fiel auf „Fedora" mit Apache-Version 2.4.18. Nach kurzer Zeit wurde aber Festgestellt, dass in der Apache-Version 2.4.20 viele Funktionen bearbeitet und ergänzt wurden (Apache Software Foundation 2016b). Das Betriebssystem „Fedora" hat aber diese Apache-Version nicht unterstützt. Dann wurde das Betriebssystem „openSUSE" unter Version "20160907" gefunden, das die benötigte Apache-Version unterstützt. Diese wurde als Server für die Tests genommen.

Die Unterstützung des HTTP/2 – Protokolls ist schon in Apache ab Version 2.4.17 integriert (Eissing 2015). Allerdings muss man beachten, dass sich nach der Installation des Apache-Servers das Modul zum HTTP/2 – Protokoll auch in der Liste aller installierten Module befindet: APACHE_MODULES="http2, *andere Module*". Alle Module sind in der Konfigurationsdatei „apache2" aufgelistet.

Das Modul „mod_http2" basiert auf der „Nghttp2"-Bibliothek, die die Funktionsweise des HTTP/2 – Protokolls implementiert (Apache Software Foundation 2016a), (Tsujikawa 2016). Dies bedeutet, dass zusätzlich noch diese Bibliothek mit einer möglichst neuen Version auf dem Server installiert werden muss.

Das HTTP/2 – Protokoll fordert die verschlüsselte Netzwerkverbindung nicht, deshalb sind zwei serverseitige Anwendungen möglich: h2(HTTP/2 – Protokoll über verschlüsselte Netzwerkverbindung - TLS) und h2c (über unverschlüsselte Netzwerkverbindung - TCP) (Apache Software Foundation 2016a). Gleichzeitig erlauben die Browser die Verwendung des HTTP/2 – Protokolls nur über TLS (Grigorik 2015, 21), (NGINX, Inc. 2015, 3). Dafür benötigt man eine TLS-Bibliothek mit der Erweiterung „Application-Layer Protocol Negotiation" (kurz: ALPN, siehe genaue Definition des ALPNs im Kapitel „3.3 Vorstellung des HTTP/2 – Protokolls"). Soweit bei der TLS-Bibliothek das ALPN nicht unterstützt wird, wird der Client (Browser) die Kommunikation nur unter dem HTTP/1.1 – Protokoll erlauben (Eissing 2015).

Um das HTTP/2 – Protokoll unter verschlüsselter Verbindung zu erlauben, muss man „Protocols h2 http/1.1" innerhalb der Konfigurationsdatei des Protokolls einfügen. Um das HTTP/2 – Protokoll unter unverschlüsselter Verbindung zu erlauben, muss man „Protocols h2 h2c http/1.1" ebenso innerhalb der Konfigurationsdatei des HTTP/2 – Protokolls einfügen. Alternativ können diese Regeln auch innerhalb des <VirtualHost> geschrieben werden (Eissing 2015).

All diese Schritte müssen gemacht werden, um das HTTP/2 – Protokoll anwenden zu können. Nun sollen noch ein paar Worte über die Konfiguration des Apache-Servers unter dem Betriebssystem „OpenSuse" geschrieben werden.

Die Apache Konfigurationsdateien befinden an zwei Orten:

- /etc/sysconfig/apache2
  In dieser Datei befindet sich die globale Konfiguration des Apache-Servers, wie Module; zusätzliche Konfigurationsdateien zum Einschließen; Flags, mit denen der Server gestartet werden muss und die in der Kommandozeile hinzugefügt werden müssen (Suse 2013).
- /etc/apache2/…
  In diesem Ordner befinden sich alle Konfigurationsdateien des Apache-Servers. Jeder Ordner und jede Datei beinhaltet mehrere Konfigurationsoptionen (Suse 2013).

Manche Ordner und Dateien des Ordners „apache2" unter dem Pfad „/etc/apache2" werden detaillierter dargestellt. Die Datei „httpd.conf" ist die Hauptkonfigurationsdatei des Apache-Servers, die hauptsächlich globale Einstellungen einbezieht. Es wird nicht empfohlen, um Änderungen innerhalb dieser Datei durchzuführen (Suse 2013).

Alle Host-Spezifizierten Einstellungen müssen sich innerhalb der Hostkonfiguration befinden (unter dem Ordner „/etc/apache2/vhosts.d"). Dieser Ordner beinhaltet ein Template für den Virtual Host mit und ohne SSL. Jede Datei mit dem Dateityp „.conf", die sich innerhalb dieses Ordners befindet, wird in die Apache-Konfiguration aufgenommen (Suse 2013).

Innerhalb des Ordners „conf.d" unter dem Pfad „/etc/apache2/conf.d" können sich alle Dateien zur Konfiguration einzelner Module befinden. Diese können in einem bestimmten Virtual Host

miteinbezogen werden (Suse 2013). In diesem Ordner befindet sich z.B. die Konfigurationsdatei „http2.conf" zum Modul „mod_http2" und die Konfigurationsdatei zum Clientseitigen Cache „mod_expires.conf". Diese und alle anderen Dateien in diesem Ordner wurden im Rahmen dieser Masterarbeit erstellt und konfiguriert.

Damit eine verschlüsselte TLS – Verbindung aufgebaut werden kann, muss man zuerst dem Server ein SSL – Zertifikat und den dazugehörigen Schlüssel übergeben. In die Datei zur Konfiguration des <VirtualHosts>: „vhosts.d" befinden sich zwei Pfade, einer zum SSL – Zertifikat und einer zum SSL – Schlüssel:

SSLCertificateFile /etc/apache2/ssl.crt/server.crt
SSLCertificateKeyFile /etc/apache2/ssl.key/server.key

Im Vergleich zum öffentlichen SSL – Zertifikat, müssen begrenzte Zugriffsrechte zum geheimen SSL – Schlüssel erstellt werden.

4.2. Vorbereitung zur Testevaluation

Nachdem alle zu untersuchenden Parameter von „Navigation Timing API" und „Ressource Timing API" definiert wurden, kann die Webapplikation unter unterschiedlichen Einstellungen getestet werden.

In diesem Unterkapitel wurden folgende Fragen ausgearbeitet:

- Welches Tool könnte als Hilfsmittel für die Testausführung dienen?
- Anwendung des „Webpagetest.org" – Tools.
- Wie werden die zur Untersuchung ausgewählten Parameter berechnet?
- Wie werden die Tests ausgewertet?
- Wie kann die Funktionsweise des HTTP/2 – Protokolls am lokalen PC untersucht werden?

Welches Tool könnte als Hilfsmittel für die Testausführung dienen?

Wie im Kapitel „2. Der kritische Rendering – Pfad und seine wichtigsten Komponenten" erwähnt wurde, verfügt fast jeder moderne Browser sowohl über die „Navigation Timing API" als auch über die „Ressource Timing API". Deshalb ist es möglich, die zur Untersuchung ausgewählten Parameter am lokalen PC aufzunehmen. Neben den Parametern aus der „Navigation Timing API" und der „Ressource Timing API" ist auch ein Messwert wichtig, der den Zeitpunkt misst, zu dem erste Pixel auf dem Viewport erscheinen. Es wäre auch interessant, einen visuellen Vergleich zu machen, mittels dem die Reihenfolge der geladenen Ressourcen zu sehen ist. Dafür können sogenannte „Screenshots" dienen, die in der Größe des Viewports das Laden der Webapplikation aufnehmen. Dadurch entsteht eine Reihe von Bildern, die das visuelle Laden der Webapplikation veranschaulicht.

Wenn man in den Tests zusätzlich die Parameter unter mobilen Netzwerkverbindungen untersuchen möchte, gibt es bei manchen Browsern die Möglichkeit, um die Parameter für die Bandbreite zum Hoch- und Herunterladen und die Latenz einzustellen. Dadurch wird eine bestimmte Netzwerkverbindung simuliert. Meistens befinden sich diese Einstellungen in den Entwicklertools des Browsers.

Allerdings muss man beachten, dass nicht jeder Browser alle diese notwendigen Funktionen besitzt. Zum Beispiel sind im Browser „Mozilla Firefox" die Funktionen zur Aufnahme der Screenshots und der Simulation der Netzwerkverbindung nicht verfügbar. Um zuverlässigere Ergebnisse zu bekommen, müssen die Parameter mehrmals getestet werden. Aus dem Browser am lokalen PC kann man die Parameter der „Navigation Timing API" und der „Ressource Timing API" in der *.har Datei speichern. Diese kann zwar ausgewertet werden, aber bei mehreren Tests würde die Übertragung der einzelnen Parameter zu viel Zeit in Anspruch nehmen.

Deshalb wurde entschieden, das online Tool: „Webpagetest.org" (WebPagetest 2016a) zu benutzen. Dieses ist ein sehr umfangreiches Tool für die Web – Performance Analyse, das über viele Funktionen verfügt, die für die clientseitige Performance – Optimierung hilfreich sein können. Mit dem „Webpagetest.org" – Tool ist es möglich, um alle im Kapitel „2. Der kritische Rendering – Pfad und seine wichtigsten Komponenten" beschriebenen Parameter zu untersuchen. Unter anderem kann man mithilfe des „Webpagetest.org" – Tools (Webpagetest 2016a) mit bestimmten Parametern die Webapplikation mehrmals in einem Browser aufrufen. Wenn die Tests fertig sind, werden alle Informationen zum Ressourcenwasserfall, der „Navigation Timing API", der „Ressource Timing API" und der „*.har" Datei für den gesamten Zeitraum zur Verfügung gestellt. Außerdem kann man aus den aufgenommenen Screenshots sehen, wann und was im Browserfester angezeigt wird.

Einstellungen des „Webpagetest.org" – Tools

Bei dem „Webpagetest.org" – Tool gibt es eine große Auswahl an Einstellungen. Zur Standardeinstellung gehören die Lokalisierung und der Browser, aus denen die Webapplikation aufgerufen werden soll. Im Rahmen dieser Masterarbeit wurde entschieden, die Webapplikation vom Ort „Frankfurt, Germany" aufzurufen. An diesem Ort stehen drei Browser zur Verfügung: „Mozilla Firefox" mit Version 47.0, „Google Chrome" mit Version 52.0 und „Microsoft IE11". Innerhalb des „Webpagetest.org" – Tools im Bereich „Advanced Settings" im ersten Tab „Test Settings" kann man die Netzwerkverbindung auswählen (es gibt mehrere mit der Standardkonfiguration) oder man kann eine neue hinzufügen und diese zum Testen benutzen. Im Rahmen dieser Masterarbeit wurde entschieden, die Tests bei einer Kabel-Netzwerkverbindung unter Standardkonfigurationen zu belassen. Diese hat folgende Parameter:

Bandbreite zum Herunterladen: 5000 Kbit/s
Bandbreite zum Hochladen: 1000 Kbit/s
Latenz: 28 ms

Um ausgewählte Parameter und das Verhalten der Webapplikation beim Laden genauer untersuchen zu können, wurde entschieden, die Webapplikation auch unter mobilen Netzwerken zu testen.

Eine entscheidende Rolle für das Laden der Webseite spielt die Latenz. Es wurden Performance – Tests gemacht, in denen untersucht wurde, wie groß der Einfluss der Bandbreite und der Latenz auf die Gesamtladezeit („Load Time") der Webseite ist (Abb. 19). Im ersten Test aus der Abb. 19 wird die Latenz fixiert und die Bandbreite wurde von 1 Mbps bis 10 Mbps inkrementell erhöht. Man sieht, dass sich die Ladezeit der Webseite bis 5 Mbps ändert. Danach spielt die Bandbreite kaum eine Rolle mehr bei der Ladezeit. Anders ist die Situation mit der Latenz, mit deren Zunahme auch die Ladezeit proportional wächst (Grigorik 2013, 177).

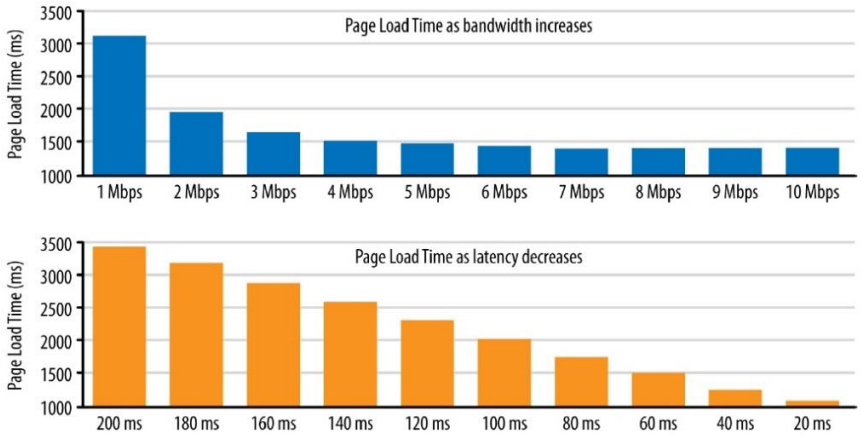

Abb. 19: Gesamtladezeit der Webseite unter Einfluss von Bandbreite (oben) und Latenz (unten). (Grigorik 2013, 177)

Da die mobile 3G – Netzwerkverbindung über eine große Latenz verfügt, wurde entschieden, diese für die Untersuchungen unter mobilen Netzwerken zu nehmen. Deren Werte wurden innerhalb Deutschlands genauer untersucht. In Deutschland gibt es vier Netzbetreiber: Telekom, Vodafone, E-Plus und O2. Es werden die Werte für die Bandbreite zum Hoch- und Herunterladen und die Latenz der einzelnen Netzbetreiber ermittelt. Für die Parameteruntersuchung unter 3G-Netzwerken wurde dann der Mittelwert von allen Werten der Netzbetreiber genommen und innerhalb des „Webpagetest.org" – Tools eingestellt.

Werte für 3G-Netzwerkverbindung innerhalb Deutschlands wurden nach OpenSignal (2016a) in Tabelle 4 dargestellt.

Provider	Bandbreite zum Herunterladen	Bandbreite zum Hochladen	Latenz
Telekom (OpenSignal 2016a)	3,7 Mb/s	0,9 Mb/s	338 ms
Vodafone (OpenSignal 2016a)	4,8 Mb/s	1,5 Mb/s	263 ms
E-Plus (OpenSignal 2016c)	4,4 Mb/s	1,2 Mb/s	251 ms
O2 (OpenSignal 2016d)	3,7 Mb/s	1,2 Mb/s	377 ms
Mittelwert	4,5 Mb/s	1,2 Mb/s	307,25 ms

Tab. 4: 3G- Netzwerkverbindungen in Deutschland, Stand: 25.07.2016

Nachdem die Netzwerkverbindung innerhalb des „Webpagetest.org" – Tools eingestellt wird, kann festgelegt werden, wie viele Tests innerhalb eines Durchgangs gemacht werden sollen (max. 9 Webseitenaufrufe pro ein Durchgang möglich). Im Rahmen dieser Masterarbeit wurde entschieden, für jede Untersuchung drei Durchläufe mit jeweils neun Tests zu machen. D.h., dass für jeden zu untersuchenden Fall die Webapplikation 27 Mal unter gleichen Bedingungen aufgerufen wurde.

Beim nächsten Parameter des „Webpagetest.org" – Tools kann eingestellt werden, ob die Webapplikation bei jedem Test nur ein Mal (First View) oder zwei Mal (Repeat View) aufgerufen werden soll. Beim zweiten Aufruf sieht man, welche Ressourcen nicht im Browser Cache geladen wurden.

Wenn man beim nächsten Parameter die Option „Capture Video" auswählt, kann man nachdem der Testdurchgang fertig ist, Screenshots aus dem simulierten Viewport anschauen.

Beim Parameter „Label" kann man den Namen für den Testdurchgang vergeben.

Es gibt noch viele Einstellungen, die innerhalb dieses Tools angewendet werden können. Oben wurden nun alle Parameter beschrieben, die für die Tests in dieser Masterarbeit relevant sind.

Woher kann man die für die Auswertung benötigten Parameter bekommen?

Nachdem der Test mithilfe des „Webpagetest.org" - Tools durchgeführt wurde, hat man die Möglichkeit, um die aufgenommenen Parameter zu speichern. Es gibt drei Dateien, die dafür zur Verfügung stehen: „Page Data.csv", die alle Parameter der „Navigation Timing API" beinhaltet; „Object Data.csv", die alle Parameter der „Ressource Timing API" beinhaltet und „HTTP Archive (.har)", die alle Informationen über HTTP-Transaktionen gespeichert hat. Soweit man vor dem Test den Testparameter „Capture Video" ausgewählt hat, hat man die Möglichkeit bei der Auswahl eines einzelnen Tests (z.B. Run 1) die aufgenommenen Screenshots zusammen mit dem Ressourcenwasserfall auf einer Timeline zu beobachten. Alle aufgenommenen

Screenshots kann man auch speichern. Außerdem besteht die Möglichkeit, auf der Basis von Screenshots ein Video generieren zu lassen.

Zusätzlich kann man die durchgeführten Tests innerhalb des „Webpagetest.org" – Tools miteinander vergleichen. Dies ist sehr nützlich, wenn man mehrere Einsätze miteinander vergleichen möchte (z.B. Einsatz mit und ohne „Server Push"). Dadurch kann man Unterschiede innerhalb des Ressourcenwasserfalls und der aufgenommenen Screenshots deutlicher sehen.

Alle oben beschriebenen Dateien wurden für jeden durchgeführten Test gespeichert und sind auf dem beigelegten Datenträger einsehbar. Um die Testdurchgänge mit ihren dazugehörigen Tests unterscheiden zu können, sind die Dateien mit den Ziffern von 1 bis 3 markiert.

Wie werden die zu untersuchenden Parameter berechnet?

Außer den Parametern der „Navigation Timing API" und der „Ressource Timing API" verfügen die vom „Webpagetest.org" – Tool erstellten Dateien „Page Data.csv" und „Object Data.csv" noch über weitere Zusatzinformationen. Manchmal speichert das „Webpagetest.org" – Tool zusätzliche Metriken, die auch für die Untersuchung der Ladezeit der Webapplikation hilfreich sind. Z.B. kann man im Dokument „Page Data.csv" den Parameter „Start Render" finden, der ausgegeben wird, sobald erste Pixel auf dem Viewport erscheinen werden. Dieser Parameter ist eine eigene Metrik des „Webpagetest.org" - Tools (Viscomi et al. 2014, 15).

„Load Time" ist auch eine eigene Metrik des „Webpagetest.org" - Tools. Diese misst die Zeit, zwischen dem ersten an den Server gesendeten Request und der Ausgabe des „load"-Events durch den Browser. Diese Metrik ist auch als „domComplete"-Parameter aus der „Navigation Timing API" bekannt (Viscomi et al. 2014, 11).

Wenn man den Parameter „Time To First Byte" aus „Page Data.csv" und „Object Data.csv" vergleicht, ist zu sehen, dass diese sich um einige Millisekunden unterscheiden. Für die Testauswertung wird der Parameter aus „Object Data.csv" genommen, weil der Parameter „Time To First Byte" in dieser Datei für jede einzelne Ressource (inkl. HTML Datei) gespeichert ist.

„Bytes Out" wurde als Metrik weder in der Dokumentation des „Webpagetest.org" – Tools (WebPagetest 2016b) noch bei Viscomi et al. 2014 beschrieben. Aber für den Parameter „Bytes In" heißt es, dass dieser zeigt, wie viele Bytes innerhalb der Sitzung der Client vom Server bekommen hat (Viscomi et al. 2014, 11). Deshalb kann man davon ausgehen, dass der Parameter „Bytes Out" anzeigt, wie viele Bytes vom Client gesendet werden, um Ressourcen anzufragen. Diese Metrik wird vom „Object Data.csv" für einzelne Dateien genommen, um zu schauen, wie viele Anfragebytes ausgegeben werden. Diese Metrik könnte Interessant sein, wenn der Einsatz von „Server Push" getestet wird.

Der Parameter „domInteractive" ist nur innerhalb der Datei „HTTP Archive (.har)" zu treffen. Um die DOM-Erstellung berechnen zu können, wird die Differenz zwischen dem Parameter „domInteractive" und dem Parameter „Start Time" genommen. Der Parameter „Start Time" befindet

sich innerhalb der Datei „Object Data.csv“ und gibt für jede einzelne Ressource die Zeit an, zu der der Request gesendet wurde.

„Visual Progress“ ist eigentlich kein Parameter. Dies ist eine Grafik, die zeigt, wie schnell die Inhalte der Webapplikation (prozentual gesehen) auf dem Viewport erscheinen. Um den „Visual Progress“ sehen zu können, muss man ein oder mehrere Tests online innerhalb des „Webpagetest.org“ – Tools miteinander vergleichen. Alle Tests müssen mithilfe der Funktion „Capture Video“ (siehe Testparameter) aufgenommen werden. Um die Tests miteinander zu vergleichen, muss man innerhalb des „Webpagetest.org“ – Tools die URL „https://www.webpagetest.org/video/compare.php?tests=“ gefolgt von der Test-ID, die bei jedem Test in der Adressleiste des Browsers angezeigt wird, eingeben.

Wie werden die Tests ausgewertet?

Für jeden Test wird die Webapplikation 27 Mal aufgerufen. Für die Auswertung der aufgenommenen Parameter muss ein Mittel gefunden werden, um diese visuell darstellen zu können. Die erste Darstellungsidee war die tabellarische Darstellung mit drei Werten zu allen aufgenommenen Werten: der Medianwert, das Minimum und das Maximum. Diese drei Werte können als aussagekräftigste betrachtet werden. Da die Werte aus mehreren Tests miteinander verglichen werden sollen, wird sich die Darstellungsart dieser Werte in der Form eines Liniendiagramms besser eignen. In allen Tests gibt es nur eine Achse, nämlich die Zeit, an der die Parameter ausgegeben werden.

Allerdings gibt es oftmals Werte, die sich von allen anderen stark unterscheiden und an denen festgestellt werden kann, dass diese seltene Ausnahmen sind. Dann wird in solchen Fällen die gesamte Messreihe nicht mehr aussagekräftig sein, da Maximum und Minimum verzerrt sind.

Aus diesem Grund muss eine Darstellung gefunden werden, die deutlich zeigt, in welchem Bereich sich die meisten Werte befinden, wo die Ausnahmen sind, wo minimale und maximale Werte sind und wo sich der Median befindet. Der Median wurde dem arithmetischen Mittelwert vorgezogen, weil er robuster gegenüber Ausreißern ist. Für alle diese Anforderungen eignet sich das Boxplot-Diagramm gut.

Das Boxplot-Diagramm enthält ein Rechteck, die sogenannte Box, und zwei Linien, die dieses Rechteck verlängern. Die Bezeichnung für diese Linien ist „Antenne“ oder „Whisker“. Innerhalb der Box befindet sich ein Strich, der die Position des Median markiert. Abb. 20 veranschaulicht das Boxplot-Diagramm. Die Box enthält die mittleren 50% aller Werte. Der Median teilt das komplette Boxplot-Diagramm in zwei Hälften, sodass links und rechts vom Median sich jeweils 50% aller Werte befinden. Je nachdem, wo sich der Median innerhalb der Box befindet, kann man sagen, wie sich die Werte innerhalb des gesamten Zeitraums verteilen. Wenn sich der Median in der rechten Hälfte der Box befindet, ist die Datenverteilung linksschief (Abb. 20).

Wenn der Median in der linken Hälfte der Box liegt, ist die Datenverteilung rechtsschief (Wikipedia 2016a).

Die Grenzen der Box sind das untere und obere Quartil. Die Länge der Box wird durch die Differenz zwischen dem unteren und oberen Quartil definiert. Die Spannweite ist der komplette Wertebereich aller Daten. Das Maximum steht für den größten Datenwert der ganzen Spannweite. Das Minimum steht für den kleinsten Datenwert der gesamten Spannweite. Der Bereich zwischen Minimum und unterem Quartil, als auch der Bereich zwischen dem oberen Quartil und dem Maximum umfasst 25% aller Werte (Wikipedia 2016a).

Mithilfe der Antennen werden die Werte angezeigt, die sich außerhalb der Box befinden. Die Länge der Antenne wird als das maximal 1,5-Fache der Länge der Box bestimmt. Dies gilt sowohl für die untere als auch für die obere Antenne. Die Antenne endet nicht genau nach der 1,5-fachen Länge der Box, sondern nimmt den Wert an, der gerade noch innerhalb der berechneten Länge liegt. Deshalb kann es sein, dass die untere und obere Antenne unterschiedlich lang sind (Wikipedia 2016a).

Alle Werte, die außerhalb der Antennen liegen, sind die Ausreißer. Die Ausreißer werden mit Kreißen im Boxplot-Diagramm dargestellt (Abb. 20). Sowohl Maximum als auch Minimum können Ausreißer sein (Wikipedia 2016a). Meistens unterscheiden sich die Ausreißer von allen anderen Werten der Werteverteilung und sind ein Zeichen dafür, dass entweder diese Werte falsch gemessen wurden oder sie eine Ausnahme darstellen. Deshalb bedeutet die Markierung als Ausreißer, dass diese Werte innerhalb der 27 gemessenen Werte nicht betrachtet werden sollten.

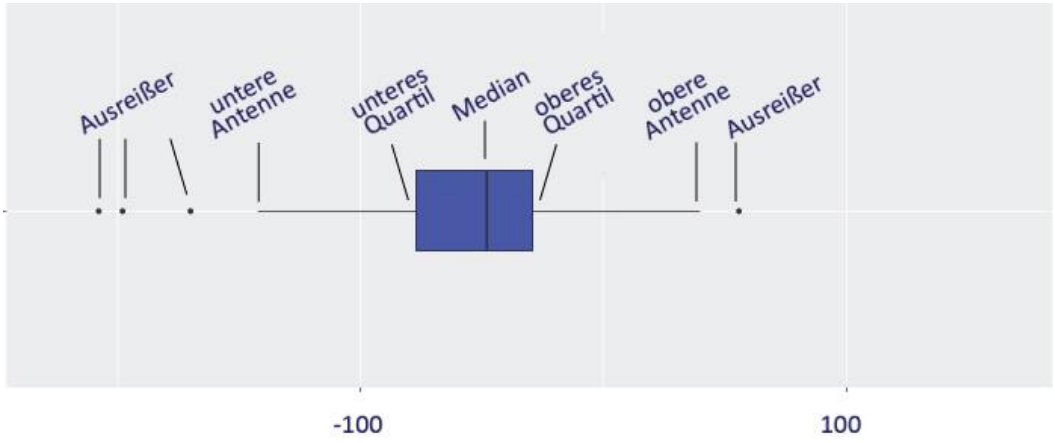

Abb. 20: Beispiel eines Boxplot-Diagramms

Die für die Tests verwendeten Boxplot-Diagramme wurden mithilfe der Statistikprogrammiersprache „R", der Entwicklungsumgebung „RStudio" und der Erweiterung „ggplot2" erstellt (The R Foundation 2016), (Wickham 2013), (RStudio 2016).

Wie kann man die Funktionsweise des HTTP/2 – Protokolls am lokalen PC untersuchen?

Für eine detaillierte Untersuchung des Netzwerk-Datenverkehrs von Datenprotokollen eignet sich die „Wireshark" Software sehr gut (Wireshark Foundation 2016a). Diese wird für Untersuchungen am lokalen PC benutzt.

Es wird der TCP-Port 443 für das HTTP/2 – Protokoll benutzt (Wireshark Foundation 2016b) weil die Browser dieses Protokoll nur unter verschlüsselter TLS-Verbindung laufen lassen (Grigorik 2015, 21). Da es sich um eine verschlüsselte Verbindung handelt, kann „Wireshark" nicht ohne zusätzliche Mittel die Kommunikation zwischen Client und Server betrachten (Shaver 2015). Um verschlüsselte Verbindungen unter dem HTTP/2 – Protokoll mithilfe von „Wireshark" untersuchen zu können, muss man das HTTP/2 – Protokoll innerhalb von „Wireshark" entschlüsseln (Wireshark Foundation 2016b). Um eine Verschlüsselte Verbindung knacken zu können, kann ein „Man-in-the-Middle" – Angriff benutzt werden. Der „Man-in-the-Middle" stellt sich zwischen beide Kommunikationspartner und überwacht den kompletten Datenverkehr zwischen den Netzwerkteilnehmern. Dies wird ermöglicht, indem sich der Angreifer bei beiden Kommunikationspartnern für den jeweils anderen Partner ausgibt. Die Informationen, die der Angreifer bekommt, können eingesehen und manipuliert werden (Wikipedia 2016b).

Die Browser „Mozilla Firefox" und „Google Chrome" ermöglichen das mitschreiben des symmetrischen Sitzungsschlüssels. Dieser wird benutzt, um den TLS- Datenverkehr in einer Datei verschlüsselt abzulegen. Wenn nun „Wireshark" diese Datei übergeben wird, kann der TLS-Datenverkehr eingesehen werden (Shaver 2015).

Der lokale PC, der für die Testuntersuchung benutzt wird, hat das Betriebssystem „Microsoft Windows 7". Um den zuvor beschriebenen „Man-in-the-Middle-Angriff" bei den Browsern „Mozilla Firefox" und „Google Chrome" benutzen zu können, muss man eine Umgebungsvariable im Betriebssystem setzen. Der Name der Variable muss „SSLKEYLOGFILE" sein und der Wert der Variable „C:\Users*Name des Benutzers*\sslkeylog.log" (Shaver 2015). Die Datei muss unter diesem Pfad existieren.

Im zweiten Schritt muss man diese Datei an „Wireshark" übergeben. Man braucht dafür „Wireshark" ab der Version 1.12.0 (CloudFlare 2016). Unter „Edit"->„Preferences" ->„Protocols" ->„SSL" im Feld „(Pre)-Master-Secret log filename" muss man den Pfad zur Datei „sslkeylog.log" übergeben.

Diese Methode hilft, um in „Wireshark" die Inhalte der TCP-Pakete (unter Decrypted SSL Data) und die Inhalte des HTTP/2 – Protokolls („Frames" mit allen Parametern) zu sehen. Man sollte nur nicht vergessen, dass das HTTP/2 – Protokoll ein binäres Protokoll ist, d.h., um alle Inhalte verstehen zu können, müssen die binären Inhalte erst entschlüsselt werden.

Um aus dem gesamten Datenmitschnitt nur das HTTP/2 – Protokoll verfolgen zu können, muss man den Filter „http2" anwenden. Es werden alle Pakete angezeigt, die zu diesem Protokoll

gehören. Es gibt eine Liste von Filtern, die man für den Datenmitschnitt aus „Wireshark“ anwenden kann (Wireshark Foundation 2016c). Es ist auch möglich, um Filter selbst zu erstellen, wenn man z.B. einen konkreten „Stream“, „Frame“ oder eine Datei verfolgen möchte. Wenn die Benutzeroberfläche von „Wireshark“ benutzt wird, muss man einen bestimmten Parameter (z.B. „Stream Identifier: 7“) auswählen, dann aus dem Kontextmenü „Apply as Filter“ „Selected“ auswählen.

5. Tests und Testevaluation

In diesem Kapitel wird der praktische Teil der Masterarbeit vorgestellt. Innerhalb dieses Kapitels werden mehrere Fragen in Bezug auf die Funktionsweise des HTTP/2 – Protokolls und dessen Funktion „Server Push" beantwortet. Es werden mehrere Tests zu clientseitigen Techniken der Performance – Optimierung durchgeführt und ausgewertet. Außerdem werden Tests zum Vergleich der Ladezeiten bei den HTTP/1.1- und HTTP/2 – Protokollen durchgeführt.

5.1. Wie funktioniert „Server Push"?

Die Funktionsweise des „Server Pushes" wurde im theoretischen Teil der Masterarbeit kurz erläutert. Allerdings stellen sich einige Fragen, wie diese Funktion des neuen Protokolls tatsächlich funktioniert. Einige davon sind:

- Wird die Lieferung der HTML Datei verzögert, bis alle gepushten Ressourcen den Client erreicht haben? Wenn nein, wie wird der „Server Push" – Einsatz genau funktionieren? Wann werden diese Ressourcen zum Client geliefert?
- In welcher Reihenfolge werden die gepushten Ressourcen beim Client ankommen?
- Lassen sich die unterschiedlichen Typen von gepushten Ressourcen priorisieren?
- Werden sich die gepushten Dateien bei erneutem Aufruf der Webapplikation im Browser Cache befinden?

In diesem Teil wird die genauere Funktionsweise des „Server Pushes" bei einem Apache Server mit Version 2.4.20 beschrieben. Um einzelne Frames während des Transports besser zu beobachten, kann man mithilfe von „Wireshark" ein Blick in die einzelnen Pakete geworfen werden. Als Client wird der Browser „Mozilla Firefox" mit Version 47.0 genommen, damit die Versionen des „Webpagetest.org" – Tools mit der Version des lokalen PCs gleich bleiben werden. Es werden zusätzliche Tests gemacht, in denen das HTTP/2 – Protokoll in anderen Webbrowsern untersucht wird.

Wie kann man „Server Push" mit Apache Server anwenden?

Für „Server Push" ist die „H2Push"-Direktive verantwortlich. Diese ist im Apache-Modul „mod_http2" schon enthalten. Diese Direktive muss sich innerhalb des <VirtualHost> befinden. „Server Push" ist standardmäßig eingeschaltet. Um „Server Push" auszuschalten, muss man innerhalb des <VirtualHost> „H2Push off" setzen. Der „Server Push"-Ablauf wird gemessen, indem man die „Link-Header" der Responses untersucht. Wenn ein Link mit dem Attribut „rel=preload" versehen ist, wird er als zu pushende Ressource betrachtet. „Link-Header" können entweder im Code der Webapplikation stehen oder mithilfe von „mod_headers" konfiguriert werden. Um die zweite Variante anwenden zu können, muss beachtet werden, dass das Apache-Modul „mod_headers" installiert ist (Apache Software Foundation 2016a).

„Link-Headers“ können folgendermaßen angewendet werden:

```
<Location /index.html>
    Header add Link "</css/style.css>;rel=preload"
    Header add Link "</css/slider.css>;rel=preload"
</Location>
```

(Apache Software Foundation 2016a)

Man muss allerdings beachten, dass die Ressourcen nur einmal aufgelistet sind, weil es keine Überprüfung nach Duplikaten innerhalb dieses Moduls gibt. Falls die Ressourcen mehrmals aufgelistet werden, werden diese mehrmals zum Client gepusht (Apache Software Foundation 2016a).

Es gibt eine wichtige Voraussetzung bei der Verwendung des „Server Pushes“: dieser funktioniert nur dann, wenn der Client signalisiert, dass er gepushte Ressourcen annehmen will. Die meisten Browser machen dies, aber manche, wie „Safari 9“, akzeptieren sie nicht. „Server Push“ funktioniert nur für Ressourcen, die zu demselben <VirtualHost> gehören, wie die angefragte Ressource (Apache Software Foundation 2016a).

Alle Ressourcen, die per „Header“ im „<VirtualHost>“-Bereich des Apache-Servers aufgelistet sind, müssen auch im Code der Webapplikation stehen.

Im Fall des „Server Pushes“ werden alle „Streams“ vom Server geöffnet. In der Dokumentation des Apache-Moduls „mod_http2“ steht, dass sich verschiedene Typen von gepushten Ressourcen je nach MIME-Typ unterschiedlich priorisieren lassen. Das Ziel des Priorisierens besteht darin, die verfügbare Bandbreite zwischen den Ressourcen sinnvoll aufzuteilen. Z.B. ist es wichtig, dass die CSS Dateien so schnell wie möglich den Client erreichen. Deshalb ist für sie eine hohe Priorität sehr wichtig. Wie die serverseitige Priorisierung eingestellt wird, ist dem Webentwickler überlassen. Man kann mit Servereinstellungen Gewichte für „Streams“, je nach MIME-Typ der Dateien, bestimmen (Apache Software Foundation 2016a).

Wie funktioniert die serverseitige Priorisierung der per „Server Push“ übergebenen Ressourcen?

Serverseitige Priorisierungen lassen sich mithilfe der „H2PushPriority“-Direktive implementieren (Apache Software Foundation 2016a). Diese Funktion ist dann besonders hilfreich, wenn unterschiedliche Typen von Ressourcen per „Server Push“ übergegeben werden. Wenn z.B. sowohl kritische CSS Dateien als auch kritische JavaScript Dateien gepusht werden müssen, ist es hilfreich, zuerst alle CSS Dateien an den Client zu liefern, damit es keine zusätzlichen Verzögerungen geben wird.

Normalerweise werden alle „Streams“ clientseitig geöffnet und diese werden vom Client mithilfe von PRIORITY – Frames priorisiert. Aber im Fall von gepushten Ressourcen, deren

„Streams“ serverseitig geöffnet werden, darf der Server entscheiden, wie Prioritäten gesetzt werden sollen (Apache Software Foundation 2016a).

Die „H2PushPriority“-Direktive gibt die Möglichkeit, die Gewichte der gepushten Ressourcen zu steuern. Diese kann sich entweder innerhalb des <VirtualHost> oder innerhalb der HTTP/2 – Serverkonfiguration befinden. Der Syntax der „H2PushPritority“-Direktive ist der Folgende:

H2PushPriority *mime-type* (after | before | interleaved) (weight)

Wenn man gleiche Gewichte auf alle Ressourcentypen anwenden möchte, muss man anstatt eines bestimmten MIME-Typs an der Stelle *„mime-type*“ einen „*“ schreiben. Je größer das Gewicht ist, desto höher wird die Priorität gesetzt. „After“, „Before“ und „Interleaved“ zeigen, wie die Bandbreite zwischen den gepushten Ressourcen und dem clientseitigen „Stream“ (von dem die gepushten Ressourcen abhängig sind) aufgeteilt werden soll. „After“ bedeutet, dass die gepushten Ressourcen nach dem clientseitigen „Stream“ geschickt werden müssen. „Interleaved“ bedeutet vermischt mit dem clientseitigen „Stream“. „Before“ steht dafür, dass gepushte Ressourcen den Client vor dem clientseitigen „Stream“ erreichen (Apache Software Foundation 2016a).

Die standardmäßige Priorisierungsregel lautet: *H2PushPriority * After 16*. Dies bedeutet, dass alle per „Server Push“ übergebenen Ressourcentypen, die vom clientseitigen „Stream“ abhängig sind, nach diesem „Stream“ mit einem Gewicht von 16 gesendet werden (Apache Software Foundation 2016a).

Um genauer zu zeigen, wie Gewichte eingesetzt werden müssen, wird ein Beispiel gezeigt. Es wird angenommen, dass X der clientseitige „Stream“ ist, der vom Y-“Stream“ abhängig ist. Der Server muss entscheiden, wie er die „Streams“ P1 und P2 zum X-“Stream“ pushen wird. Wenn die Regel „H2PushPriority text/css Interleaved 256" lautet, kann der Server es so interpretieren, dass alle gepushten CSS Ressourcen die gleiche Abhängigkeit und das gleiche Gewicht haben werden, wie der clientseitige „Stream“. Das effektive Gewicht („Effective weight“, kurz: „ew“) wird mit einer Formel berechnet:

$P1_{ew} = X_w * (P1_w/256)$, $P1_{ew}$ - effektives Gewicht des gepushten „Streams“
X_w - Gewicht des clientseitigen „Streams“
$P1_w$ - eingestelltes Gewicht der gepushten Ressource

(Apache Software Foundation 2016a)

Das effektive Gewicht darf nicht mehr als 256 sein. Im aktuellen Beispiel, wenn die gepushte Datei eine CSS Ressource ist und das Gewicht auf 256 gesetzt wurde, wird der gepushte „Stream“ ein effektives Gewicht gleich dem Gewicht des X-“Streams“ haben. Dies bedeutet, wenn X und P1 Daten zum Senden haben werden, wird P1 genau so viel Bandbreite bekommen, wie der clientseitige „Stream“ (in diesem Fall X-„Stream“). Wenn das Gewicht der gepushten

Ressource auf 512 eingestellt würde, würde er doppelt so viel Bandbreite bekommen, als der clientseitige „Stream". Falls das Gewicht der gepushten Ressource auf 128 eingestellt würde, würde er nur die Hälfte der verfügbaren Bandbreite nach dem clientseitigen „Stream" bekommen (Apache Software Foundation 2016a).

Fallbeispiel: Funktionsweise des „Server Pushs"

Es wird die Startseite der Test-Webapplikation mit 10 gepushten CSS Dateien untersucht. Es wurden keine serverseitigen Priorisierungen für sie gesetzt. Etwas später wird untersucht, wie weit gepushte Ressourcen die Webapplikation beeinflussen können, wenn diese serverseitig priorisiert werden. Nach der Aufnahme des Datentransportes mit „Wireshark" kann man alle HTTP/2 – Frames genau betrachten, die zwischen Client und Server ausgetauscht werden. Die Webapplikation wird mithilfe des Browsers „Mozilla Firefox" aufgerufen. Um diese ohne Cache aufrufen zu können, muss man mit der Tastenkombination „Strg + F5" die Seite aktualisieren.

In der „Wireshark"-Aufnahme sieht man, dass der Client kein besonderes Signal gibt, die Ressourcen annehmen zu wollen, wie es in der Dokumentation des „mod_http2"-Moduls des Apache-Servers steht (Apache Software Foundation 2016a).

Unmittelbar nachdem die HTML Datei vom Server aufgerufen wurde, kommen alle PUSH_PROMISE – Frames der gepushten Dateien zum Client. Da nur zehn Dateien gepusht werden, passen alle „Frames" in ein TCP – Paket. PUSH_PROMISE – Frames kommen genau in der Reihenfolge, wie sie im <VirtualHost> beim Server aufgelistet sind. Jeder PUSH_PROMISE – Frame enthält Informationen über den „Stream", zu dem er gehört (im aktuellen Fall ist es die HTML Datei mit dem „Stream" Identifier 13); über die eigene gepushte „Stream-ID" („Promised-Stream-ID") und über den „Frame Header". Es gibt eine wichtige Bemerkung: „Promised-Stream-ID" wird nur einmal pro PUSH_PROMISE – Frame gesetzt. Danach werden gewöhnliche „Stream Identifier" verwendet. Es ist zu beachten, dass alle „Promised-Stream-IDs" gerade Zahlen haben, weil deren „Streams" serverseitig geöffnet werden.

Nachdem alle PUSH_PROMISE – Frames den Client erreicht haben, werden die dazugehörigen „Streams" vom Client priorisiert. Allerdings ist schwer zu erklären, warum vom Server geöffnete „Streams" vom Client priorisiert werden. Dies wiederspricht der Dokumentation des Moduls „mod_http2" des Apache-Servers (Apache Software Foundation 2016a).

Der Client schickt die PRIORITY – Frames für jeden einzelnen gepushten „Stream". Diese „Frames" haben eine Reihenfolge gemäß des „FIFO"-Prinzips: wenn der PUSH_PROMISE – Frame des „Streams" „A" den Client zuerst erreicht hat, wird der PRIORITY – Frame dieses „Streams" als erster gesendet. Danach erreichen per „Server Push" übergebene Ressourcen den Client genau in der Reihenfolge, wie diese innerhalb des <VirtualHost> aufgelistet sind.

Die wichtigsten Informationen, die die PRIORITY – Frames enthalten, sind „Stream Identifier" (ID des gepushten „Streams"), „Stream Dependency" und „Weight". Alle PRIORITY – Frames

haben gleiche Gewichte. Dies könnte daran liegen, dass die Ressourcen keine bestimmten serverseitigen Priorisierung haben.

Nachdem alle PRIORITY – Frames geschickt wurden, erreichen den Client HEADERS – Frames gefolgt vom DATA – Frame der HTML Datei. Nach der Beobachtung werden alle PUSH_PROMISE- und PRIORITY – Frames dem Client geliefert, bevor der DATA – Frame der HTML Datei den Client erreicht. Danach werden alle gepushten Ressourcen den Client erreichen. Erst nachdem alle per „Server Push" übergebenen Ressourcen den Client erreichen, werden weitere Ressourcen der Webapplikation vom Server abgefragt.

Werden die Ressourcen nach dem zweiten Aufruf der Webapplikation im Browser Cache landen?

Diese Fragestellung wurde am lokalen PC geprüft und es wurde festgestellt, dass bei der Verwendung des für den Test festgelegten Browsers „Mozilla Firefox" die gepushten Ressourcen wieder neu geladen werden. Dies wurde mithilfe von „Wireshark" und den Entwicklertools des Browsers festgestellt. Es wäre jedoch zu erwarten gewesen, dass, nachdem der PUSH_PROMISE – Frame den Client erreicht hat, der dazugehörige „Stream" vom Client abgelehnt worden wäre. Vermutlich hätte an dieser Stelle ein RST_STREAM – Frame vom Client gesendet werden sollen.

Bei der Verwendung des „Webpagetest.org" – Tools konnte dagegen festgestellt werden, dass alle Ressourcen der Webapplikation beim zweiten Aufruf aus dem Browser Cache geladen werden. Dabei waren die Browserversionen des „Webpagetest.org" – Tools und des lokalen PCs die gleichen.

5.2. Zwischenfazit: Wie funktioniert „Server Push"

Wofür könnte „Server Push" hilfreich sein? Welche Parameter des kritischen Rendering – Pfades können durch „Server Push" verzögert werden?

Nachdem genauer betrachtet wurde, wie der „Server Push" – Einsatz funktioniert, wurden viele der am Anfang gestellten Fragen beantwortet. Es werden keine kompletten gepushten Dateien zum Client geliefert, bevor die HTML Datei den Client erreicht. Stattdessen werden, bevor der DATA – Frame der HTML Datei den Client erreicht, PUSH_PROMISE – Frames des gepushten „Streams" zum Client geliefert und dann werden Prioritäten mithilfe von PRIORITY – Frames gesetzt. Deshalb wird die HTML Datei mit jeder gepushten Ressource durch PUSH_PROMISE- und PRIORITY – Frames später an den Client geliefert. Deshalb wird mit dem „Server Push" – Einsatz der Parameter „Time To First Byte" und möglicherweise die DOM-Erstellung verzögert. Andererseits werden Bytes für Requests dieser Dateien gespart, sodass sie schneller den Client erreichen werden. Deshalb wird der Render – Baum vermutlich schneller aufgebaut, wenn die für die Erstdarstellung kritischen Ressourcen per „Server Push" übergeben werden. Außerdem ist zu erwarten, dass die Gesamtladezeit für alle Ressourcen durch weniger angefragte Bytes verkürzt wird.

Wie viele Dateien können per „Server Push" übergegeben werden?

Wenn viele Dateien gepusht werden, ist zu erwarten, dass möglicherweise der Parameter „Time To First Byte" und die DOM-Erstellung deutlich verzögert werden. Deshalb könnte dieser Fall die Erstellung des Render – Baums bremsen.

Wichtig scheint auch zu sein, dass keine großen Ressourcen per „Server Push" übergeben werden. Wenn diese lange Zeit zum Herunterladen brauchen werden (dies ist der Fall in mobilen Netzwerken), dann werden alle anderen Ressourcen auf der Seite warten, bis diese Dateien vollständig heruntergeladen wurden.

Wann und in welcher Reihenfolge werden gepushte Ressourcen zum Client geliefert?

Gepushte Ressourcen werden den Client erst erreichen, nachdem die HTML Datei vollständig geladen wurde. Wenn alle gepushten Ressourcen gleich priorisiert werden, werden sie nacheinander geliefert. Im Falle, dass es keine serverseitige Priorisierung gibt, ist die Reihenfolge der Auflistung im <VirtualHost>-Bereich sehr wichtig. Dies bedeutet, wenn im <VirtualHost>-Bereich des Apache-Servers die Dateien z.B. in der Reihenfolge CSS Dateien, Webschrift und dann wieder CSS Dateien aufgelistet sind, dann werden die letzteren CSS Dateien warten, bis die Webschriften vollständig geladen wurden. Diese Situation muss man vermeiden.

Außerdem sollen alle kritischen Ressourcen mitgepusht werden, auch wenn es notwendig ist, um nicht-kritische Ressourcen (z.B. Webschriften oder Bilder) per „Server Push" zu übergeben. In diesem Fall muss man dafür sorgen, dass zuerst alle kritischen Dateien den Client erreichen werden. Dadurch wird vermieden, dass die Erstellung des Rendering – Baums verzögert wird.

Lassen sich die unterschiedlichen Typen von gepushten Ressourcen priorisieren?

Innerhalb des Moduls „mod_http2" des Apache Servers wurde die Direktive der serverseitigen Priorisierung von gepushten Ressourcen implementiert. Deshalb können gepuschte Ressourcen mit verschiedenen MIME-Typen unterschiedlich priorisiert werden.

Im zuvor dargestellten Fallbeispiel wurde die serverseitige Priorisierung der gepushten Ressourcen ausgeschaltet. Dennoch werden PRIORITY – Frames vom Client an den Server geliefert. Dies wiederspricht der Dokumentation des „mod_http2" Moduls (Apache Software Foundation 2016a).

Es werden später Tests gemacht und untersucht, wie die Prioritäten für unterschiedliche Typen von gepushten Ressourcen funktionieren.

Werden per „Server Push" übergebene Dateien im Browser Cache landen?

Die Ergebnisse des „Webpagetest.org" – Tools sind unterschiedlich zu den Ergebnissen, die am lokalen PC gemacht wurden. Die Browserversionen sind die gleichen. Im Browser des „Webpagetest.org" – Tools sieht man, dass beim zweiten Aufruf der Webapplikation alle Dateien bereits im Browser Cache sind. Dies betrifft auch alle gepushten Ressourcen. Als die Webapplikation am lokalen PC aufgerufen wurde, wurden beim zweiten Aufruf per „Server Push" übergebene Dateien noch Mal vom Server geladen.

Die Funktionsweise des „Server Pushs", die oben beschrieben wurde, bezieht sich auf einen konkreten Apache Server mit der Version 2.4.20. Als Client wurde der Browser „Mozilla Firefox" mit der Version 47.0 ausgewählt. Dies bedeutet, dass sich das oben beschriebene Verhalten auf eine konkrete Version der Serverimplementierung und des oben gewählten Browsers bezieht und sich demnach möglicherweise unter anderen Versionen/Implementierungen anders verhalten wird. In weiteren Tests wurden Fälle untersucht, bei denen andere Browser als Client gewählt wurden.

5.3. Serverseitige Priorisierung von per „Server Push" übergebenen Ressourcen

Oben wurde die Funktionsweise serverseitiger Priorisierungen von per „Server Push" übergebenen Ressourcen beschrieben. Es ist interessant zu schauen, wie diese tatsächlich funktionieren. Deshalb wurden zwei Tests zur „H2PushPriority"-Direktive durchgeführt und ausgewertet.

5.3.1. Test 1: Priorisierung von per „Server Push" übergebenen CSS Dateien und Schriften

Testbedingungen: untersucht wurde die Startseite der Webapplikation. Es wurden alle für die Seite benötigten CSS Dateien (10 Stück) und fünf nicht-kritische Schrift Dateien per „Server Push" übergegeben. Die Ressourcen wurden im <VirtualHost> folgendermaßen aufgelistet: alle CSS Dateien, danach alle für die Startseite benötigten Schriften.

Damit gepushte „Streams" die HTML Datei nicht verzögern, wird der Parameter „After" verwendet. In diesem experimentellen Fall wurden in der HTTP/2 – Serverkonfiguration innerhalb der „H2PushPriority"-Direktive für CSS Dateien und Schriften die gleichen Gewichte übergegeben, um zu schauen, wie die Bandbreite zwischen den „Streams" aufgeteilt wird:

H2PushPriority text/css after 256
H2PushPriority application/x-font-woff after 256 (Behnke 2013)
H2PushPriority application/x-font-truetype after 256 (Behnke 2013)

Mit den in diesem Test eingestellten Gewichten ist zu erwarten, dass die Frames von CSS Dateien und Schriften die verfügbare Bandbreite gleichmäßig zwischen sich aufteilen werden. Dies

bedeutet, dass sich die „Frames" der „Streams" von CSS Dateien und Schriften während der Lieferung zum Client regelmäßig abwechseln sollten.

Die Webapplikation wurde am lokalen PC mithilfe des Browsers „Mozilla Firefox" mit Version 47.0 aufgerufen. Die Reihenfolge und die Prioritäten von einzelnen Frames wurden mithilfe von „Wireshark" genauer untersucht.

Testergebnisse: aus dem Datenmittschnitt in „Wireshark" sieht man, dass PUSH_PROMISE – Frames genau in der Reihenfolge zum Client geliefert werden, wie die dazugehörigen Ressourcen im <VirtualHost> gelistet sind. Nach diesem Moment ist zu erwarten, dass entweder der Server die Prioritäten zum Client schicken wird oder der Client mithilfe von PRIORITY – Frames unterschiedliche Gewichte für die „Streams" der Webschriften und der CSS Dateien setzen wird. Sehr erstaunlich ist, dass alle vom Client gesendeten PRIORITY – Frames das gleiche Gewicht (Weight: 1) haben. Dies passiert wahrscheinlich, weil die Gewichte für CSS Dateien und Schriften in der HTTP/2 – Serverkonfigurationsdatei gleich eingestellt wurden.

Nachdem fast alle PRIORITY – Frames geliefert wurden, kommt der HEADERS – Frame, gefolgt vom DATA – Frame des „Streams" der HTML Datei (siehe die Datei „Mitschnitt Startseite.pcapng"). Danach erreicht den Client der letzte PRIORITY – Frame des letzten „Streams" der gepushten Ressource.

Wie früher erwähnt wurde, werden HEADERS- und DATA – Frames von „Streams" der gepushten Ressourcen den Client in der FIFO-Reihenfolge erreichen. Es ist eher zu erwarten, dass Frames der gepushten Ressourcen durchgemischt wurden, weil die Bandbreite zwischen CSS Dateien und Schriften gleichmäßig aufgeteilt wurde.

Noch zu bemerken ist, dass innerhalb des HEADERS – Frames der gepushten Dateien, die vom Server an den Client geschickt werden, steht, dass diese keine Prioritäten haben. Dies kann man innerhalb des „Flags=Priority: false" sehen. Dies signalisiert, dass keine Abhängigkeiten und Gewichte für den aktuellen „Stream" gesetzt wurden (Belshe et al. 2015, 34).

Zusammenfassung: Priorisierung von per „Server Push" übergebenen CSS Dateien und Schriften.

Aus den Ergebnissen des aktuellen Tests kann man sagen, dass in diesem Fall die Priorisierungen der gepushten Ressourcen, die serverseitig eingestellt wurden, nicht funktioniert haben. Per „Server Push" übergebene Ressourcen werden letzten Endes clientseitig priorisiert. Trotz der Erwartungen, werden „Streams" der gepushten Ressourcen während der Lieferung nicht miteinander vermischt.

Alle gepushten Ressourcen erreichen den Client genau in der Reihenfolge, in der sie innerhalb des <VirtualHost> aufgelistet wurden. Die Einstellungen in der „H2PushPriority"-Direktive haben in diesem Fall nichts verändert. Es wird im nächsten Test untersucht, ob die Prioritäten mit anderen Einstellungen besser funktionieren.

Außerdem ist es interessant zu betrachten, wie die Ressourcenreihenfolge in den Entwicklertools des Browsers aussieht. Aus dem Ressourcenwasserfall sieht man, dass die CSS Dateien als erste den Client erreichen. Gleichzeitig sieht man, dass die per „Server Push" übergebenen Schriften ganz am Ende geladen werden. Aus dem Datenmitschnitt von „Wireshark" ist jedoch ersichtlich, dass dies nicht der Fall ist und Schriften gleich nach den CSS Dateien ankommen. Dies könnte bedeuten, dass die Implementierung der Entwicklertools in „Mozilla Firefox" für die „Server Push"-Technologie noch nicht bereit ist.

5.3.2. Test 2: Priorisierung von per „Server Push" übergebenen CSS Dateien und Bildern

Testbedingungen: untersucht wurde die Dozenten-Seite (dozentent2.html) der Webapplikation. Es wurden alle für die Seite benötigten CSS Dateien (acht Stück) und 10 nicht-kritische kleine Bilder per „Server Push" übergeben. Die Ressourcen wurden im <VirtualHost> folgendermaßen aufgelistet: zuerst alle CSS Dateien, danach die Bilder.

Damit gepushte „Streams" die HTML Datei nicht verzögern werden, wird der Parameter „After" verwendet. In diesem Beispiel wurden innerhalb der HTTP/2 – Serverkonfiguration für CSS Dateien und Bilder unterschiedliche Gewichte übergegeben. Um genau zu schauen, wie Prioritäten funktionieren, wird ein simulierter Fall verwendet, in dem alle „.jpg" Dateien höhere Priorität haben, als die CSS Dateien:

```
H2PushPriority text/css          after  256
H2PushPriority image/jpeg        after 512
```

Mit den eingestellten Gewichten in diesem Test ist zu erwarten, dass alle „Frames" von „Streams" der *.jpg – Ressourcen als erste den Client erreichen werden. Danach sollten alle CSS Dateien dem Client zugestellt werden.

Die Webapplikation wurde am lokalen PC mithilfe des Browsers „Mozilla Firefox" mit Version 47.0 aufgerufen. Die Lieferreihenfolge der „Streams" der gepushten Ressourcen und die Prioritäten der einzelnen „Frames" wurde mithilfe von „Wireshark" genauer untersucht.

*Testergebnisse***:** aus dem Datenmittschnitt von „Wireshark" sieht man, dass PUSH_PROMISE Frames genau in der Reihenfolge zum Client geliefert werden, wie die dazugehörigen Dateien im <VirtualHost> gelistet wurden. Danach, wie im früheren Test, setzt der Browser die Prioritäten für alle vom Server geöffneten „Streams". Alle vom Browser erzeugten PRIORITY – Frames haben erstaunlicherweise das gleiche Gewicht. Nach dem 13. PRIORITY – Frame wird der HEADERS – Frame der HTML Datei den Client erreichen. Danach kommen die restlichen PRIORITY – Frames gefolgt vom DATA – Frame der HTML Datei (siehe die Datei „Mitschnitt Dozenten2.pcapng").

Interessant zu bemerken ist, dass zuerst alle CSS Dateien den Client erreichen, obwohl höhere Prioritäten in der HTTP/2 – Serverkonfigurationsdatei für „.jpg“ Dateien gesetzt wurden.

Bei der Lieferung stimmt die Reihenfolge der gepushten Bilder ab dem 22. „Stream“ nicht mehr ganz. Nachdem der DATA – Frame für die 22. „Stream“-Id zugestellt wurde, werden die HEADERS – Frames zuerst für den 34. bis 38. „Stream“ zugestellt und dann für den 24. bis 30 „Stream“ an den Client geliefert (siehe die Dateien auf dem beigelegten Datenträger). Die DATA – Frames behalten diese Reihenfolge. Leider ändert sich diese Vorgehensweise nicht sehr: CSS Dateien wurden trotzdem zuerst dem Client zugestellt.

Genauso wie im vorherigen Test, wurde bemerkt, dass innerhalb des HEADERS – Frames der gepushten Dateien steht, dass diese keine Prioritäten haben. Dies kann man innerhalb des „Flags=Priority: false“ sehen. Dies bestätigt, dass serverseitige Priorisierungen nicht funktioniert haben.

Zusammenfassung: Priorisierung von per „Server Push“ übergebenen CSS Dateien und Bildern.

Die Ergebnisse des aktuellen Tests sind zu den Ergebnissen des früheren Tests sehr ähnlich. Priorisierungen werden clientseitig erstellt. Außerdem haben diese das gleiche Gewicht, obwohl dieses serverseitig unterschiedlich eingestellt wurde.

Auch dieses Beispiel hat gezeigt, dass die Einstellungen in der „H2PushPriority“-Direktive an der Ressourcenreihenfolge der gepushten Ressourcen nicht viel ändert. Dafür wurde noch keine Erklärung gefunden. Es könnte entweder daran liegen, dass sich das „mod_http2“-Modul noch im experimentalen Stadium befindet (Apache Software Foundation 2016a) und nicht immer funktioniert, wie in der Dokumentation steht. Oder es liegt evtl. daran, dass die vom Server erstellten Gewichte für „Streams“ gepushter Ressourcen vom Browser (im aktuellen Fall Mozilla Firefox, 47.0) nicht akzeptiert werden.

5.4. Zwischenfazit: Serverseitige Priorisierung von per „Server Push“ übergebenen Ressourcen

Aus den zuvor gemachen Tests kann man sagen, dass serverseitige Einstellungen in der „H2PushPriority“-Direktive keinen Einfluss auf die Priorisierung der gepushten Ressourcen haben. Erstaunlicherweise werden Prioritäten nur clientseitig gesetzt. Dazu kommt noch, dass für alle „Streams“ der gepushten Ressourcen gleiche Gewichte gesetzt werden, wenn diese unterschiedlich serverseitig eingestellt werden. Noch zu bemerken ist, dass innerhalb des HEADERS – Frames der gepushten Dateien, die vom Server an den Client geschickt werden, steht, dass diese keine Prioritäten haben.

Falls diese Einstellungen nicht innerhalb der HTTP/2 – Serverkonfigurationsdatei gemacht werden müssen, wurden noch Versuche gemacht, um die Einstellungen der „H2PushPriority“-Direktive innerhalb des <VirtualHost> zu platzieren. Dies hat aber die gleichen Ergebnisse gezeigt.

Bisher ist noch nicht klar, welche Probleme auf dem Weg zur serverseitigen Priorisierung liegen. Dies könnte sowohl auf der Server- als auch auf der Clientseite liegen, wie es im Zwischenfazit der oben durchgeführten Tests erwähnt wurde.

Jetzt stellt sich die Frage, was diese Verhaltensweise für die Servereinstellungen bedeuten. Wie müssen Ressourcen per „Server Push" übergegeben werden, wenn serverseitige Priorisierungen nicht funktionieren? Vermutlich gibt es nur einen Weg: falls mehrere unterschiedliche Ressourcentypen gepusht werden, müssen sie im <VirtualHost> genauso aufgelistet sein, dass kritische Ressourcen beim Herunterladen nicht blockiert werden. D.h., wenn CSS Dateien und Schriften gepusht werden sollen, ist es sinnvoll, zuerst alle CSS Dateien aufzulisten und erst danach die Schriften.

5.5. Untersuchungen zum „Server Push"

Zuvor wurde die Funktionsweise mit dem Apache-Server schon kurz erwähnt und es wurden Vermutungen aufgestellt, wofür „Server Push" angewendet werden kann und wie er am besten benutzt werden könnte. Um die Vermutungen zu prüfen, werden in den folgenden Tests unterschiedliche Einsätze für die Verwendung des „Server Pushs" untersucht.

Zu untersuchende Parameter:

- „Time To First Byte"
  Zeigt an, wann das erste Byte der HTML Datei den Client erreicht. Wie im früheren Test zur Untersuchung des „Server Pushes" bemerkt wurde, passieren einige Schritte zwischen Client und Server, bevor die Daten der HTML Datei den Client erreichen. Dieser Parameter könnte sich mit zunehmender Anzahl von gepushten Ressourcen verzögern.
- DOM Erstellung
  Zeigt, ob der „Server Push" Einsatz einen Einfluss auf die Erstellung des DOMs hat. Die Verzögerung der DOM-Erstellung könnte einen Einfluss auf den Parameter „Start Render" haben.
- „Start Render"
  Zeigt, zu welchem Zeitpunkt der Webbrowser anfängt, etwas im Viewport zu zeichnen.
- „Bytes Out"
  Zeigt, wie viele Bytes vom Client gesendet werden, um Ressourcen anzufragen (Requests). Da sich dieser Parameter von Test zu Test innerhalb einer Testreihe (innerhalb der durchgeführten 27 Tests) kaum unterscheidet, wird dieser nur anhand von neun durchgeführten Tests gemessen. Es gibt kaum einen Unterschied zwischen den Parametern „Bytes Out" unter Kabel- oder 3G-Verbindung.
- „domComplete"
  Zeigt, wie lange es gebraucht hat, bis alle zur Webapplikation gehörigen Ressourcen heruntergeladen wurden.

— „Visual Progress“
  Zeigt, wie schnell die Inhalte der Webapplikation (prozentual gesehen) auf dem Viewport erscheinen.

Wenn die Ergebnisse der Erstdarstellung durch den „Server Push“ – Einsatz Vorteile bringen werden, werden die Parameter „Start Render“ und „domComplete“ unter einer 3G-Verbindung zusätzlich geprüft. Es wird davon ausgegangen, dass sich der Parameter „Time To First Byte“ und die DOM-Erstellung durch den „Server Push“ – Einsatz auf gleiche Weise verhalten werden, wie unter Kabel-Verbindung. Aus diesem Grund werden sie nicht noch zusätzlich unter einer 3G-Verbindung geprüft.

Es wurden immer zwei unterschiedliche Einsätze miteinander verglichen. Fast für jeden Einsatz wurde die Webapplikation mithilfe des „Webpagetest.org“ – Tools 27 Mal aufgerufen. Alle Daten zur Parameteruntersuchung wurden heruntergeladen. Als Ausrufepunkt der Webapplikation wurde der Ort Frankfurt a.M. in Deutschland gewählt.

Es werden folgende Daten aufgenommen: „Page Data“: Daten, die die gesamten Werte der „Navigation Timing API“ liefern; „Object Data“: Daten, die die gesamten Werte der“ Resource Timing API“ liefern; „*.har“: Datei, die alle Informationen zum gesamten Navigationsraum liefert. Außerdem wurden die Screenshots vom Viewport des Browsers aufgenommen, um prüfen zu können, wie und in welcher Reihenfolge die Ressourcen auf dem Viewport erscheinen.

5.5.1. Test 1: „Server Push“ – Einsatz für kritische CSS Ressourcen

Es wird nichts im Viewport angezeigt, bevor das CSSOM nicht fertig gebaut ist. Wie schon im theoretischen Teil beschrieben wurde, muss man, um die Ladezeit der Webapplikation beschleunigen zu können, so schnell wie möglich die notwendigen CSS Ressourcen an den Client übergeben. Aus diesem Grund könnte der „Server Push“ – Einsatz sehr praktisch sein. Dadurch wird der Client Bytes für Anfragen und die dafür gebrauchte Zeit sparen. Es ist zu erwarten, dass erste Pixel im Viewport schneller gezeichnet werden, als ohne „Server Push“ mit CSS Dateien. Möglicherweise wird sich dadurch auch die Gesamtladezeit verkürzen.

Testbedingungen: untersucht wurde die Startseite der Webapplikation. Im Fall der Anwendung mit „Server Push“ wurden alle für die Seite benötigten CSS Dateien (10 Stück) per „Server Push“ übergeben. CSS und JavaScript Dateien werden für das HTTP/2 – Protokoll aufgeteilt.

Im aktuellen Test wird geprüft, wie weit die wichtigsten Parameter des kritischen Rendering – Pfades durch den Einsatz von „Server Push“ mit CSS Dateien beeinflusst werden. Außerdem wird geschaut, wie sich die Ladezeit der Webapplikation und die Darstellung im Viewport ändern werden.

„Time To First Byte“

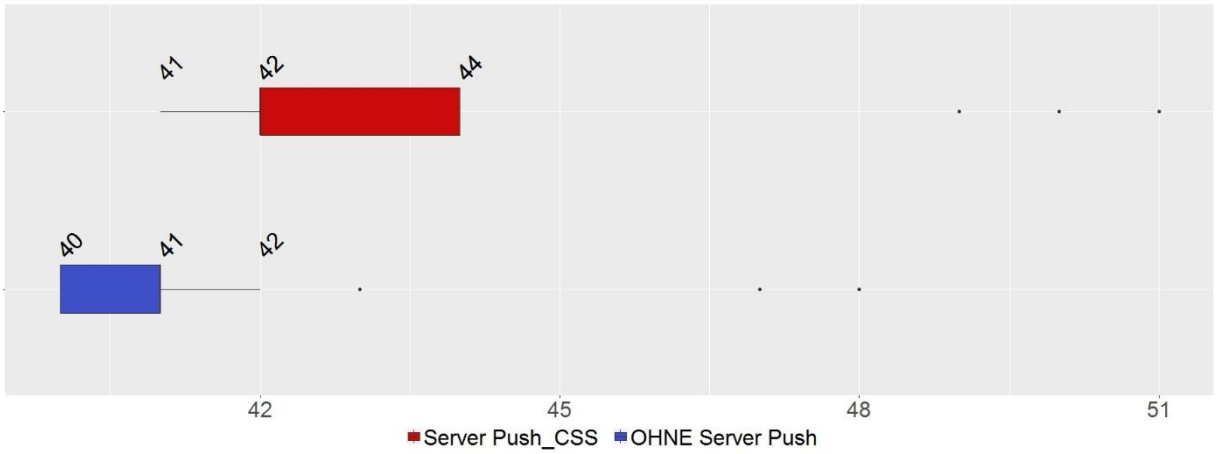

Abb. 21: Vergleich des Parameters „Time To First Byte“ mit und ohne „Server Push“ mit CSS Dateien, jeweils mit Kabel-Verbindung.

Parameter	HTTP/2: „Server Push“ mit CSS Dateien	HTTP/2: Ohne „Server Push“
Median	42 ms	41 ms
Unteres Quartil	42 ms	40 ms
Oberes Quartil	44 ms	41 ms

Tab. 5: Vergleich des Parameters „Time To First Byte“ mit und ohne „Server Push“ mit CSS Dateien, jeweils mit Kabel-Verbindung.

Aus oben dargestelltem Boxplot-Diagramm kann man erkennen, dass der Parameter „Time To First Byte“ für die Anwendung mit „Server Push“ mit CSS Dateien ein wenig später ausgeführt wird. Der Unterschied zur Anwendung ohne „Server Push“ beträgt im Durchschnitt 1 ms.

Wie zuvor vermutet wurde, wurde etwas Zeit in Anspruch genommen, um alle PUSH_PROMISE – Frames zum Client zu liefern und die „Streams“ der gepushten Ressourcen zu priorisieren. Im aktuellen Fall wird eine Kabel-Verbindung benutzt und nur 10 CSS Dateien per „Server Push“ übergeben. Aus diesem Grund ist die Verzögerung dieses Parameters nicht groß.

DOM Erstellung

Aus Abb. 22 lässt sich schließen, dass die DOM Erstellung mit „Server Push“ im Durchschnitt etwas später als bei der Anwendung ohne „Server Push“ beendet wird. Die ersten 50% der Werte der ersten Messreihe liegen zwischen 117 ms und 143 ms und der Unterschied zwischen den Medianwerten liegt bei 12 ms.

Der Grund dafür, wie zuvor bereits vermutet wurde, liegt daran, dass PUSH_PROMISE – Frames und die folgende Priorisierung etwas Zeit in Anspruch nehmen. Dies könnte die Erstellung des DOMs etwas bremsen.

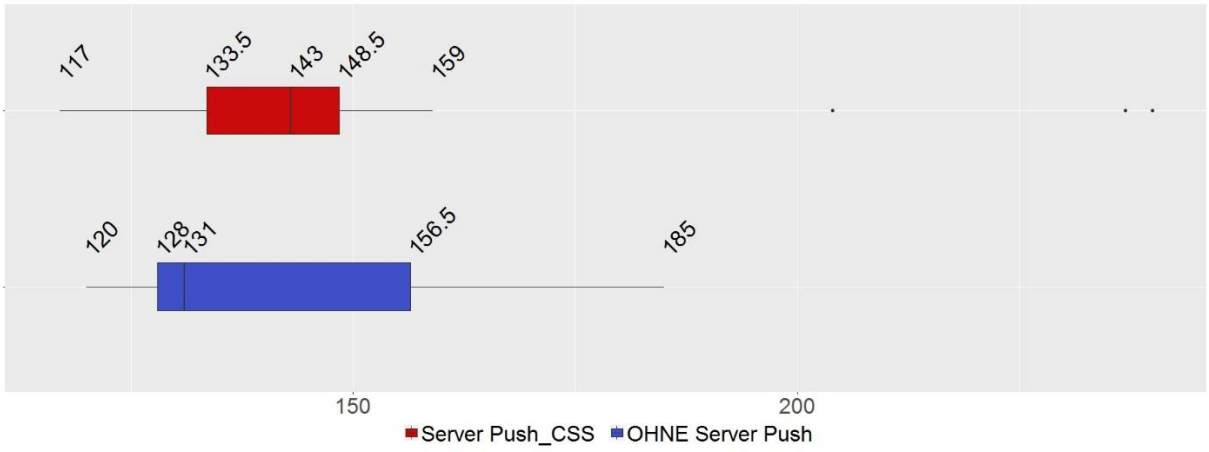

Abb. 22: Vergleich der Zeit zur DOM Erstellung unter Einsatz von „Server Push" mit CSS Dateien und ohne den Einsatz von „Server Push", jeweils mit Kabel-Verbindung.

Parameter	HTTP/2: „Server Push" mit CSS Dateien	HTTP/2: Ohne „Server Push"
Median	143 ms	131 ms
Unteres Quartil	133,5 ms	128 ms
Oberes Quartil	148,5 ms	156,5 ms

Tab. 6: Vergleich der Zeit zur DOM Erstellung unter Einsatz von „Server Push" mit CSS Dateien und ohne den Einsatz von „Server Push", jeweils mit Kabel-Verbindung.

„Start Render"

Aus Abb. 23 ist zu sehen, dass „Server Push" mit CSS Dateien einen großen Einfluss auf den „Start Render" – Parameter hat. Die Werte des Parameters bei der Anwendung mit „Server Push" werden deutlich früher ausgegeben als bei der Anwendung ohne „Server Push". Die mittleren 50% der Werte der Anwendung mit „Server Push" befinden sich zwischen 781 ms und 897,5 ms. Der Unterschied zwischen den Medianwerten liegt bei 213 ms.

Aus Abb. 24 ist ersichtlich, dass der Parameter „Start Render" unter Anwendung von „Server Push" mit CSS Dateien deutlich schneller ausgegeben wurde, als ohne „Server Push" – Einsatz. Der Unterschied zwischen den Medianwerten liegt bei 304 ms.

Je schneller DOM und CSSOM aufgebaut werden, desto schneller wird der Render – Baum fertiggestellt und desto schneller wird der Browser im Viewport erste Pixel zeichnen. Obwohl der Aufbau des DOMs etwas verzögert wird, werden im Fall der Anwendung von „Server Push" mit CSS Dateien keine Anfragen an CSS Ressourcen geschickt. Dadurch werden mehrere Bytes innerhalb einer TCP – Verbindung gespart und die Dateien erreichen den Client schneller. Im aktuellen Fall liegt der Unterschied bei 213 ms unter Kabel-Verbindung und 304 ms unter 3G-Verbindung und könnte damit eine große Rolle bei der User Experience spielen.

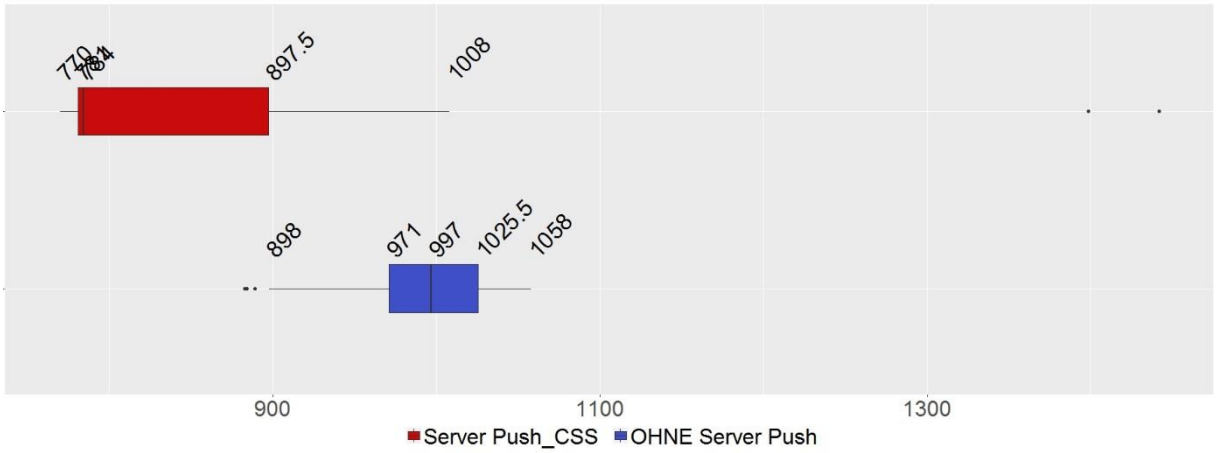

Abb. 23: Vergleich des Parameters „Start Render" beim Einsatz von „Server Push" mit CSS Dateien und ohne Anwendung von „Server Push", jeweils mit Kabel-Verbindung.

Parameter	HTTP/2: „Server Push" mit CSS Dateien	HTTP/2: Ohne „Server Push"
Median	784 ms	997 ms
Unteres Quartil	781 ms	971 ms
Oberes Quartil	897,5 ms	1025,5 ms

Tab. 7: Vergleich des Parameters „Start Render" beim Einsatz von „Server Push" mit CSS Dateien und ohne Anwendung von „Server Push", jeweils mit Kabel-Verbindung.

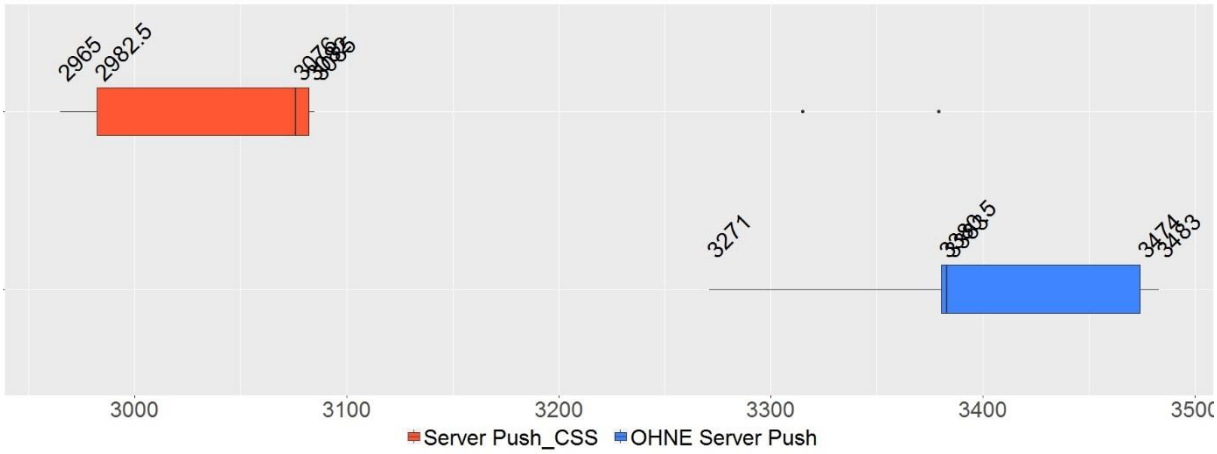

Abb. 24: Vergleich des Parameters „Start Render" bei Anwendung von „Server Push" mit CSS Dateien und ohne Anwendung von „Server Push", jeweils mit einer 3G-Verbindung.

Parameter	HTTP/2: „Server Push“ mit CSS Dateien	HTTP/2: Ohne „Server Push“
Median	3076 ms	3383 ms
Unteres Quartil	2982,5 ms	3380,5 ms
Oberes Quartil	3082 ms	3474 ms

Tab. 8: Vergleich des Parameters „Start Render“ bei Anwendung von „Server Push“ mit CSS Dateien und ohne Anwendung von „Server Push“, jeweils mit einer 3G-Verbindung.

„Bytes Out“

Mithilfe von „Server Push“ – Einsatz mit CSS Dateien ist es gelungen, um im Vergleich zu den Testreihen ohne „Server Push“ 346 Bytes pro Testreihe zu sparen.

„domComplete“

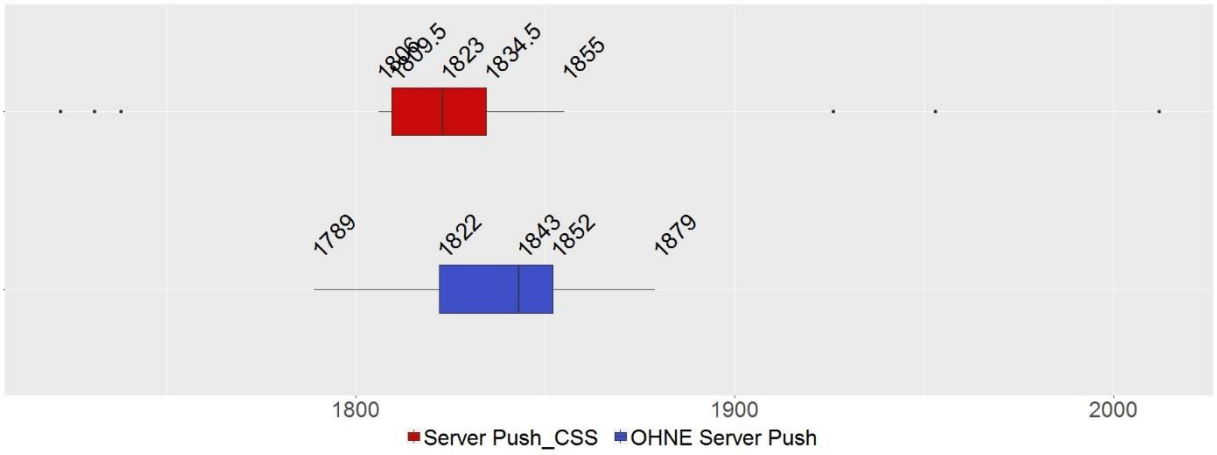

Abb. 25: Vergleich des Parameters „domComplete“ bei Anwendung von „Server Push“ mit CSS Dateien und ohne Einsatz von „Server Push“, jeweils mit Kabel-Verbindung.

Parameter	HTTP/2: „Server Push“ mit CSS Dateien	HTTP/2: Ohne „Server Push“
Median	1823 ms	1843 ms
Unteres Quartil	1809,5 ms	1822 ms
Oberes Quartil	1834,5 ms	1852 ms

Tab. 9: Vergleich des Parameters „domComplete“ bei Anwendung von „Server Push“ mit CSS Dateien und ohne Einsatz von „Server Push“, jeweils mit Kabel-Verbindung.

Aus Abb. 25 ist zu erkennen, dass die gesamte Ladezeit der HTML Datei und allen davon abhängigen Ressourcen bei der Anwendung von „Server Push“ mit CSS Dateien im Durchschnitt etwas kürzer ist, als ohne „Server Push“. Der Unterschied zwischen den Medianwerten liegt bei 20 ms.

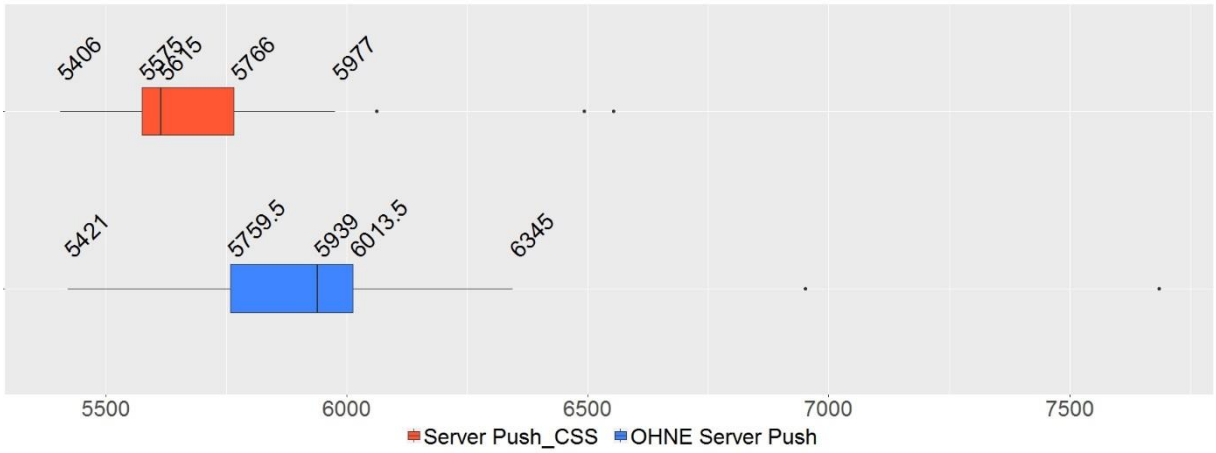

Abb. 26: Vergleich des Parameters „domComplete" bei „Server Push" - Einsatz mit CSS Dateien und ohne Anwendung von „Server Push", jeweils mit 3G-Verbindung.

Parameter	HTTP/2: „Server Push" mit CSS Dateien	HTTP/2: Ohne „Server Push"
Median	5615 ms	5939 ms
Unteres Quartil	5575 ms	5759,5 ms
Oberes Quartil	5766 ms	6013,5 ms

Tab. 10: Vergleich des Parameters „domComplete" bei „Server Push" - Einsatz mit CSS Dateien und ohne Anwendung von „Server Push", jeweils mit 3G-Verbindung.

Aus Abb. 26 ist zu sehen, dass alle Ressourcen mit „Server Push" – Einsatz unter 3G-Verbindung deutlich schneller geladen werden, als ohne „Server Push". Der Unterschied zwischen den Medianwerten beträgt 324 ms.

Man sieht, dass sowohl unter Kabel-Verbindung als auch unter 3G-Verbindung die Gesamtladezeit beim „Server Push" – Einsatz mit CSS Ressourcen schneller ist als ohne „Server Push". Die Erklärung dafür ist, dass manche Ressourcen nicht angefragt werden müssen (weil sie bereits durch den „Server Push" übergeben wurden) und die Gesamtladezeit dadurch verringert wird. Unter 3G-Verbindung wird der Unterschied zwischen den Medianwerten noch stärker.

„Visual Progress"

Aus Abb. 27 ist zu ersehen, dass mithilfe von „Server Push" erste Pixel etwas früher im Browser erscheinen werden. Allerdings sieht man, dass beide Anwendungen etwa den gleichen „Visual Progress" haben. Aus den aufgenommenen Screenshots sieht man das gleiche.

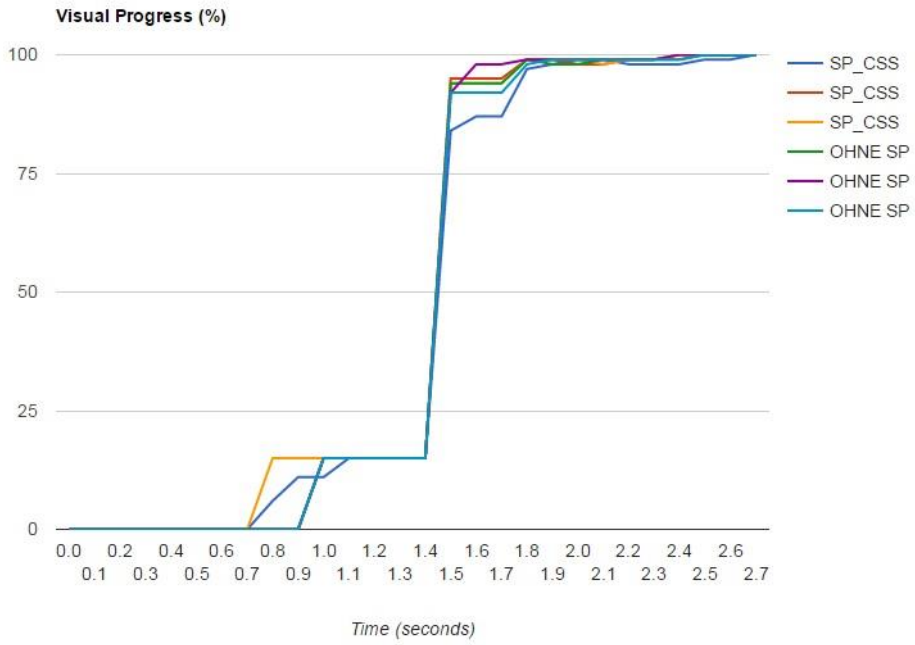

Abb. 27: „Visual Progress" bei Anwendung von „Server Push" mit CSS Dateien und bei der Anwendung ohne „Server Push" unter Kabel-Verbindung.

5.5.2. Zwischenfazit: lohnt sich die Übergabe von CSS Dateien per „Server Push" im Vergleich zur Anwendung ohne „Server Push"?

Wie man anhand der Boxplot-Diagramme zum DOM-Aufbau sehen kann, wird die Erstellung des DOMs im Fall der Anwendung von „Server Push" mit CSS Dateien etwas verzögert. Dies ist auch beim Parameter „Time To First Byte" zu beobachten. Dies passiert durch die Übergabe der PUSH_PROMISE – Frames und weiterer Streampriorisierungen von CSS Dateien. Allerdings werden mithilfe von „Server Push" 346 Anfragebytes pro Testreihe gespart. Dadurch erreichen kritische CSS Dateien den Client schneller. Aufgrund dessen, dass DOM und CSSOM parallel aufgebaut werden (Grigorik 2013, 168), wird der Render – Baum früher zur Verfügung gestellt. Als Folge werden die ersten Pixel im Viewport schneller erscheinen. Dieses Beispiel hat dies bestätigt: im aktuellen Fall unter Kabel-Verbindung beträgt der durchschnittliche Unterschied im Vergleich zur Anwendung ohne „Server Push" mehr als 1/5 Sekunde. Die Gesamtladezeit aller Ressourcen (load-Event) wird auch im Fall des Einsatzes von „Server Push" mit CSS Dateien weniger dauern.

Wenn dieser Test unter 3G-Verbindung durchgeführt wird, sieht man, dass sich die Ergebnisse im Vergleich zu den Ergebnissen mit Kabel-Verbindung noch verdeutlichen. Demnach kann gesagt werden, dass die Anwendung von „Server Push" für CSS Dateien unter mobilen Netzwerken auf jeden Fall lohnenswert ist.

5.5.3. Test 2: „Server Push" – Einsatz für nicht kritische JavaScript Ressourcen

JavaScript Dateien werden die Erstellung des DOMs und CSSOMs verlangsamen, sofern diese nicht als „nicht kritisch" durch die „defer" oder „async" Attribute markiert werden.

Es stellt sich die Frage, ob per „Server Push" übergebene nicht-kritische (mit dem Attribut „defer" oder „async" versehene) JavaScript Dateien auch einen Vorteil im Vergleich zur Anwendung ohne „Server Push" haben werden. Wenn man allerdings die Funktionsweise des „Server Pushs" betrachtet, sieht man, dass „Server Push" alle Dateien an den Client schickt und der Browser in diesem Fall keine Chance haben wird, um die Skripte nach „kritisch" oder „nicht-kritisch" zu sortieren. Die Attribute „async" und „defer" sind nur Hinweise für den Browser, die von Server-Seite ignoriert werden.

Aus diesem Grund kann man sagen, dass JavaScript Ressourcen nur dann per „Server Push" übergeben werden sollten, wenn sie tatsächlich kritisch sind. Genauso wie kritische CSS Ressourcen. Allerdings muss man entweder die Priorisierung der gepushten Ressourcen benutzen oder die Reihenfolge der Lieferung beachten, sodass JavaScript Dateien den Aufbau des CSSOM nicht blockieren werden.

5.5.4. Test 3: „Server Push" – Einsatz für kritische CSS Ressourcen und nicht kritische Ressourcen (Schriften)

Webschriften sind für die Erstdarstellung der Webapplikation nicht primär wichtig. Allerdings haben manche Webapplikationen große Überschriften mit einer besonderen Schriftart auf der Seite. Manche Browser geben den Webschriften relativ niedrige Priorisierungseinstufungen (Grigorik 2016i), deshalb werden sie als letzte von allen Ressourcen heruntergeladen. Dies bedeutet, dass sie auch ganz am Ende der Ladezeit der aufgerufenen Seite erscheinen. Es ist auch möglich, dass sie zuerst in Form einer Standard-Webschrift angezeigt werden. Erst nachdem die Original-Schriften heruntergeladen wurden, wird die Standard-Webschrift ersetzt (Grigorik 2016i). Dies könnte aus Benutzersicht einen negativen Ersteindruck machen. Die Idee des „Server Push" – Einsatzes für die Webschriften besteht darin, dass diese so schnell wie möglich auf dem Viewport erscheinen, wenn sie für den ersten Eindruck wichtig sind. Sie werden vermutlich gleich geladen, ohne dass zuerst Standard-Webschriften angezeigt werden.

Testbedingungen: untersucht wurde die Startseite der Webapplikation. Im Fall der Anwendung von „Server Push" mit CSS Dateien und Schriften wurden alle für die Seite benötigten CSS Dateien (10 Stück) und fünf Webschriften per „Server Push" übergeben. Im zweiten Fall werden nur 10 CSS Dateien per „Server Push" übergeben. Nicht alle Webschriften sind für den Ersteindruck wichtig. Es werden in diesem experimentellen Fall nun alle Schriften (fünf Dateien) übergeben, um zu schauen, wie groß der Einfluss auf den kritischen Rendering – Pfad in diesem Fall sein könnte. CSS Dateien werden für das HTTP/2 – Protokoll aufgeteilt.

Im aktuellen Test wird geprüft, wie weit die wichtigsten Parameter des kritischen Rendering – Pfades durch den Einsatz von „Server Push" für CSS Dateien und Schriften beeinflusst werden. Außerdem wird geschaut, wie sich die Ladezeit der Webapplikation und die Darstellung im Viewport ändern werden.

„Time To First Byte“

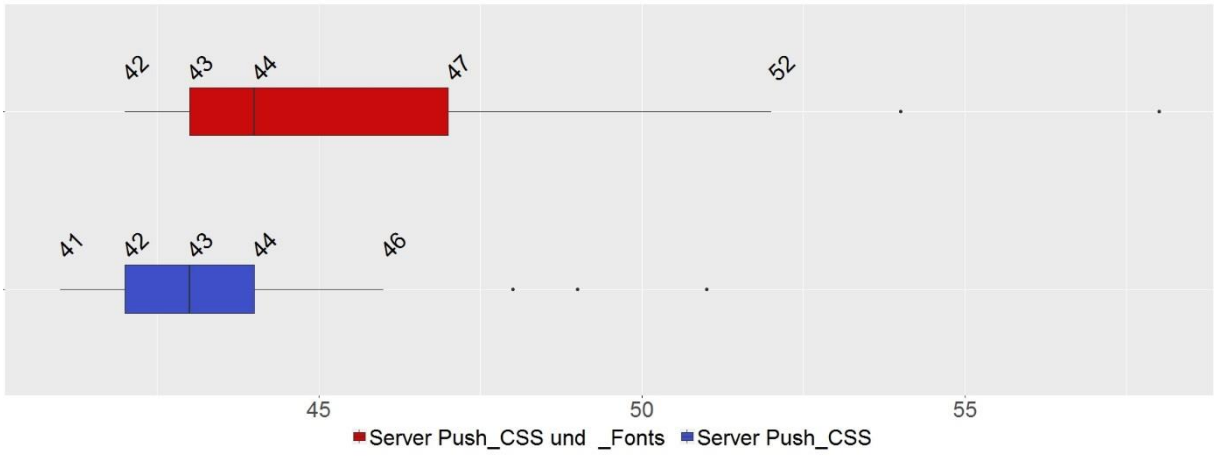

Abb. 28: Vergleich des Parameters „Time To First Byte“ 1. bei der Anwendung von „Server Push“ für CSS Dateien und Schriften und 2. bei der Anwendung von „Server Push“ nur für CSS Dateien, jeweils mit Kabel-Verbindung.

Parameter	HTTP/2: „Server Push“ mit CSS Dateien und Schriften	HTTP/2 : „Server Push“ mit CSS Dateien
Median	44 ms	43ms
Unteres Quartil	43 ms	42 ms
Oberes Quartil	47 ms	44 ms

Tab. 11: Vergleich des Parameters „Time To First Byte“ 1. bei der Anwendung von „Server Push“ für CSS Dateien und Schriften und 2. bei der Anwendung von „Server Push“ nur für CSS Dateien, jeweils mit Kabel-Verbindung.

Aus Abb. 28 ist zu sehen, dass der Unterschied zwischen den Medianwerten 1 ms beträgt. Weil das obere Quartil der ersten Messreihe im Vergleich zur zweiten Messreihe um 3 ms erhöht ist, kann man sagen, dass zusätzliche Ressourcen, die durch „Server Push“ übergeben werden, den Parameter „Time To First Byte“ im Durchschnitt etwas verzögern werden.

In diesem Fall ist der Unterschied zwischen der Anwendung von „Server Push“ für CSS Dateien und Schriften und der Anwendung von „Server Push“ nur für CSS Dateien nicht groß, weil nur fünf Schriften zusätzlich per „Server Push“ übergeben werden. Allerdings hat diese Verzögerung die zuvor beschriebenen Vermutungen bestätigt.

DOM Erstellung

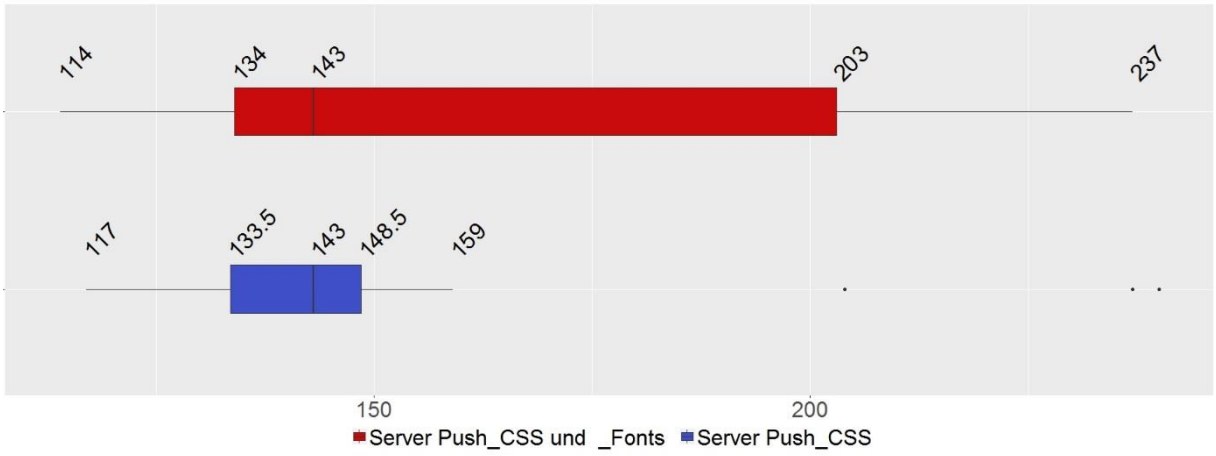

Abb. 29: Vergleich der für die Erstellung des DOMs 1. bei der Anwendung von „Server Push" für CSS Dateien und Schriften und 2. bei der Anwendung von „Server Push" nur für CSS Dateien, jeweils mit Kabel-Verbindung.

Parameter	HTTP/2: „Server Push" mit CSS Dateien und Schriften	HTTP/2: „Server Push" mit CSS Dateien
Median	143 ms	133,5 ms
Unteres Quartil	134 ms	143 ms
Oberes Quartil	203 ms	148,5 ms

Tab. 12: Vergleich der für die Erstellung des DOMs 1. bei der Anwendung von „Server Push" für CSS Dateien und Schriften und 2. bei der Anwendung von „Server Push" nur für CSS Dateien, jeweils mit Kabel-Verbindung.

Aus Abb. 29 ist zu erkennen, dass die DOM-Erstellung bei der Anwendung von „Server Push" für CSS Dateien und Schriften etwas verzögert wird. Es gibt keinen Unterschied zwischen den Medianwerten. Aber man sieht, dass das untere Quartil der ersten Messreihe bei 203 ms liegt. Deshalb kann man sagen, dass die DOM-Erstellung in etwa der Hälfte aller Fälle später fertig ist.

Diese Unterschiede liegen daran, wie zuvor vermutet wurde, dass noch fünf PUSH_PROMISE – Frames an den Client geliefert und diese Dateien noch priorisiert werden mussten. Nun ist es interessant zu beobachten, wie sehr dem Parameter „Start Render" bei der Übergabe zusätzlicher Ressourcen per „Server Push" geschadet wird.

„Start Render"

Aus Abb. 30 ist interessant zu bemerken, dass die beiden Boxplot-Diagramme ähnlich zueinander sind. Trotz der Verzögerung bei der DOM-Erstellung, wurde der Render – Baum bei An-

wendung von „Server Push“ für CSS Dateien und Schriften genauso schnell wie bei der Anwendung von „Server Push“ nur für CSS Dateien erstellt. Es gibt fast keinen Unterschied zwischen den Medianwerten.

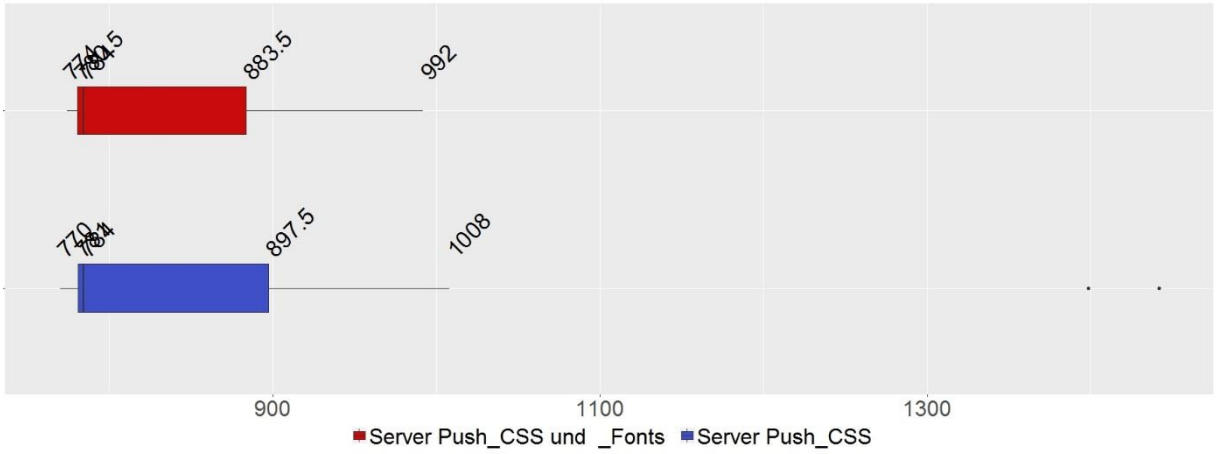

Abb. 30: Vergleich des Parameters „Start Render“ 1. bei der Anwendung von „Server Push“ für CSS Dateien und Schriften und 2. bei der Anwendung von „Server Push“ nur für CSS Dateien, jeweils mit Kabel-Verbindung.

Parameter	HTTP/2: „Server Push“ mit CSS Dateien und Schriften	HTTP/2: „Server Push“ mit CSS Dateien
Median	784,5 ms	784 ms
Unteres Quartil	780,5 ms	781 ms
Oberes Quartil	883,5 ms	897,5 ms

Tab. 13: Vergleich des Parameters „Start Render“ 1. bei der Anwendung von „Server Push“ für CSS Dateien und Schriften und 2. bei der Anwendung von „Server Push“ nur für CSS Dateien, jeweils mit Kabel-Verbindung.

Die Ergebnisse der Untersuchung unter 3G-Verbindung sind ähnlich. Aus Abb. 31 ist zu erkennen, dass die Spannweiten der beiden Messreihen vergleichbar sind. Nur wird der Median bei der Anwendung von „Server Push“ für CSS Dateien und Schriften 85 ms früher ausgegeben.

Aus diesem Beispiel, sowohl unter Kabel- als auch unter 3G-Verbindung, sieht man, dass durch die Übergabe von fünf zusätzlichen Schriftdateien, die zusammen mit allen CSS Dateien gepusht werden, der Parameter „Start Render“ nicht gestört wird. Falls mobile Netzwerkverbindungen verwendet werden, wird die Anwendung mit gepushten CSS Dateien und Schriften durchschnittlich schneller auf dem Viewport angezeigt.

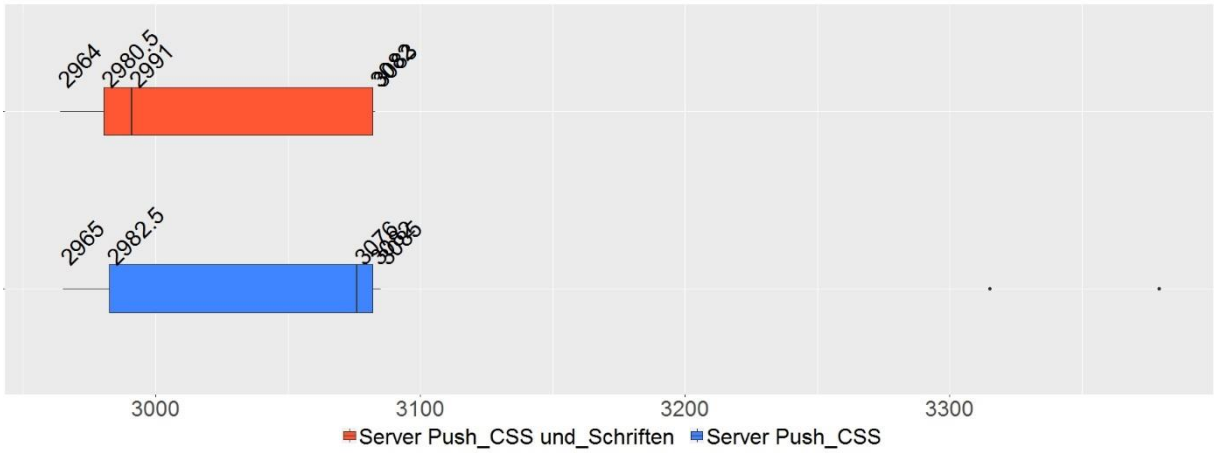

Abb. 31: Vergleich des Parameters „Start Render" 1. bei der Anwendung von „Server Push" für CSS Dateien und Schriften und 2. bei der Anwendung von „Server Push" nur für CSS Dateien, jeweils mit 3G-Verbindung.

Parameter	HTTP/2: „Server Push" mit CSS Dateien und Schriften	HTTP/2: „Server Push" mit CSS Dateien
Median	2991 ms	3076 ms
Unteres Quartil	2980,5 ms	2982,5 ms
Oberes Quartil	3082 ms	3082 ms

Tab. 14: Vergleich des Parameters „Start Render" 1. bei der Anwendung von „Server Push" für CSS Dateien und Schriften und 2. bei der Anwendung von „Server Push" nur für CSS Dateien, jeweils mit 3G-Verbindung.

„Bytes Out"

Mithilfe von „Server Push" für CSS Dateien und Schriften ist es gelungen, um im Vergleich zu den Testreihen mit „Server Push" – Einsatz nur für CSS Dateien noch 280 Bytes pro Testreihe zu sparen.

„domComplete"

In Abb. 32 ist zu sehen, dass der Parameter „domComplete" bei der Anwendung von „Server Push" für CSS Dateien und Schriften später ausgegeben wird. Der Unterschied zwischen den Medianwerten liegt bei 50 ms.

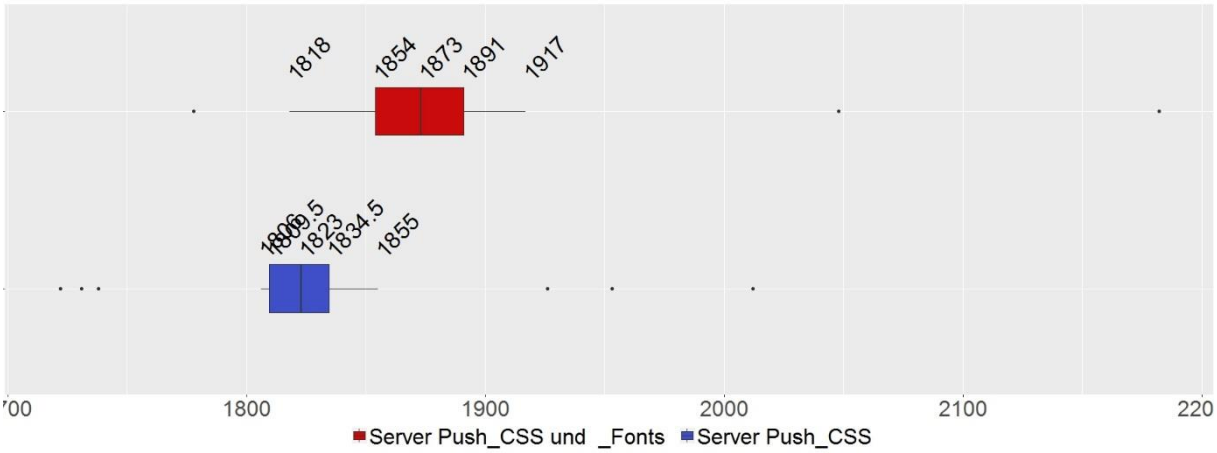

Abb. 32: Vergleich des Parameters „domComplete" 1. bei der Anwendung von „Server Push" für CSS Dateien und Schriften und 2. bei der Anwendung von „Server Push" nur für CSS Dateien, jeweils mit Kabel-Verbindung.

Parameter	HTTP/2: „Server Push" mit CSS Dateien und Schriften	HTTP/2: „Server Push" mit CSS Dateien
Median	1873 ms	1823 ms
Unteres Quartil	1854 ms	1809,5 ms
Oberes Quartil	1891 ms	1834,5 ms

Tab. 15: Vergleich des Parameters „domComplete" 1. bei der Anwendung von „Server Push" für CSS Dateien und Schriften und 2. bei der Anwendung von „Server Push" nur für CSS Dateien, jeweils mit Kabel-Verbindung.

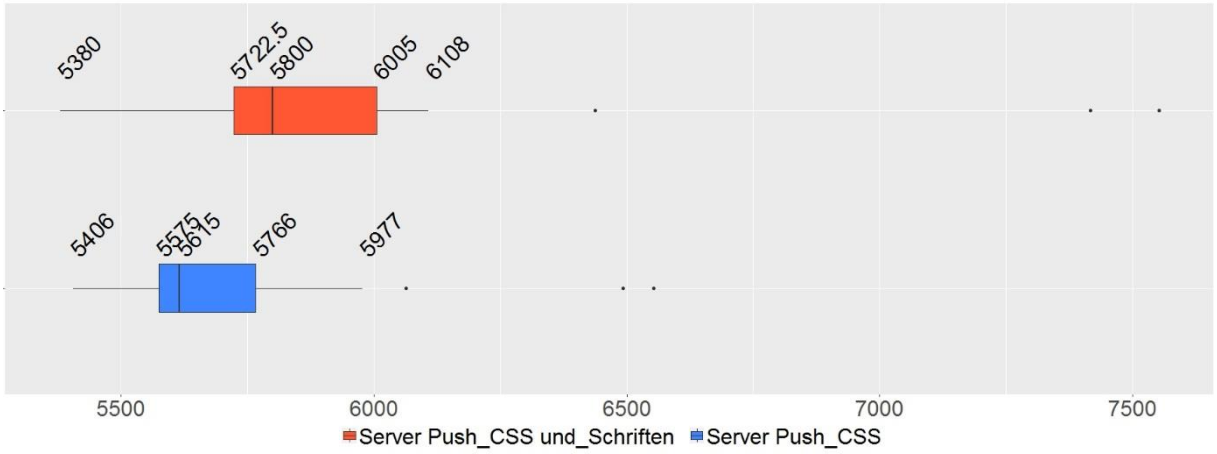

Abb. 33: Vergleich des Parameters „domComplete" 1. bei der Anwendung von „Server Push" für CSS Dateien und Schriften und 2. bei der Anwendung von „Server Push" nur für CSS Dateien, jeweils mit 3G-Verbindung.

Parameter	HTTP/2: „Server Push" mit CSS Dateien und Schriften	HTTP/2: „Server Push" mit CSS Dateien
Median	5800 ms	5615 ms
Oberes Quartil	5722,5 ms	5575 ms
Unteres Quartil	6005 ms	5766 ms

Tab. 16: Vergleich des Parameters „domComplete" 1. bei der Anwendung von „Server Push" für CSS Dateien und Schriften und 2. bei der Anwendung von „Server Push" nur für CSS Dateien, jeweils mit 3G-Verbindung.

Aus Abb. 33 sieht man, dass der Parameter „domComplete" bei der Anwendung von Server Push für CSS Dateien schneller ausgegeben wird. Der Unterschied zwischen den Medianwerten liegt bei 185 ms.

Es ist nun interessant, die aktuellen Ergebnisse mit den Ergebnissen des vorangegangenen Tests zu vergleichen, in welchem die Anwendung von „Server Push" für CSS Dateien mit der Anwendung ohne „Server Push" verglichen wurde (Test 1). In Test 1 konnte man sagen, dass der „domComplete" Parameter bei der Anwendung von „Server Push" für CSS Dateien im Mittel schneller ausgegeben wurde.

Obwohl durch „Server Push" für mehr Dateien 280 Anfragebytes gespart werden, bremst im aktuellen Fall der Zusatz von fünf Schriftdateien den „domComplete" Parameter. Der Unterschied zwischen den Medianwerten dieser Parameter hat sich unter 3G-Verbindungen noch vergrößert. Dies lässt sich nicht sofort erklären und wird in weiteren Tests geprüft.

„Visual Progress"

Aus Abb. 34 sieht man, dass die ersten Pixel bei beiden Anwendungen fast gleichzeitig erscheinen. Bei der Anwendung von „Server Push" für CSS Dateien und Schriften wird aber später der Hauptanteil (ca. 90%) des Gesamtinhaltes an den Client geliefert.

Um zu schauen, welche Inhalte tatsächlich geliefert werden, muss man einen Blick in die Screenshots werfen. Es ist zu bemerken, dass in der Anwendung von „Server Push" für CSS Dateien und Schriften die Schriften etwa 600 ms früher den Client erreichen und schon mit den ersten Pixeln angezeigt werden. Deshalb wurde das Ziel bei der Anwendung von „Server Push" für CSS Dateien und Schriften erreicht. Allerdings ist auch zu sehen, dass die Logos in der zweiten Anwendung den Client früher erreichen. Deshalb wirkt es visuell so, als ob die Anwendung nur mit gepushten CSS Dateien schneller ist. Der Grund dafür ist, dass das Herunterladen von Schriften viel Zeit braucht. Da gepushte Dateien gleich nach den HTML und CSS Dateien heruntergeladen werden, werden sie das Laden aller anderen Ressourcen auf der Seite bremsen.

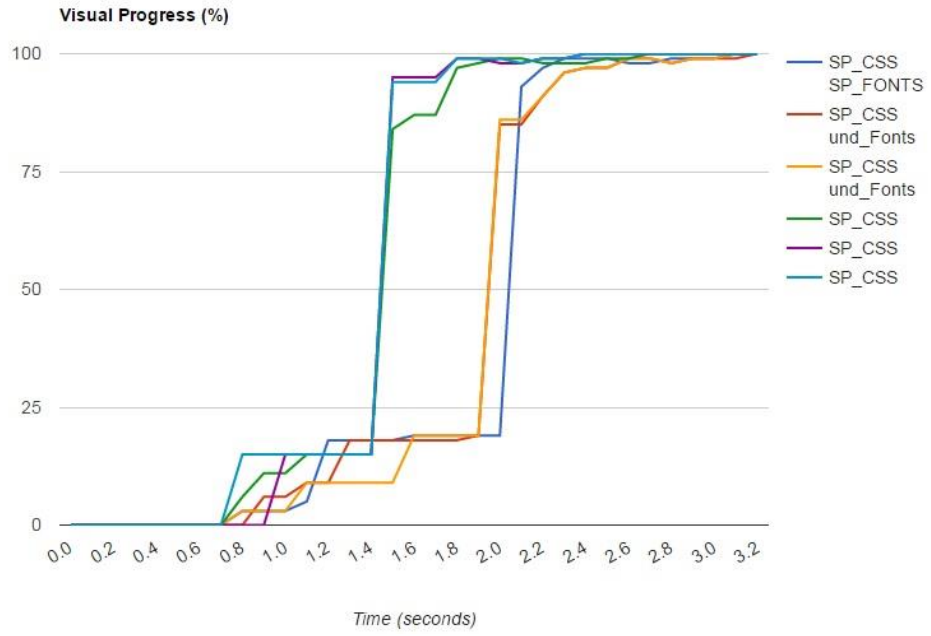

Abb. 34: „Visual Progress" 1. bei der Anwendung von „Server Push" für CSS Dateien und Schriften und 2. bei der Anwendung von „Server Push" nur für CSS Dateien, jeweils mit Kabel-Verbindung.

5.5.5. Zwischenfazit: lohnt sich die Übergabe von CSS Dateien und Schriften per „Server Push" im Vergleich zur Anwendung von „Server Push" nur für CSS Dateien?

Denkbar sind Beispiele von Webapplikationen, bei welchen die Schriften eine große Rolle bei der Erstdarstellung spielen und daher so schnell wie möglich angezeigt werden müssen. Die „Server Push" – Anwendung kann dafür geeignet sein. Im aktuellen Fall, wenn alle CSS Dateien mit übergeben werden, kann sich der Einsatz von „Server Push" für CSS Dateien und Schriften lohnen. Der Parameter „Time To First Byte" wird etwas verzögert, weil noch fünf zusätzliche PUSH_PROMISE – Frames mitübergeben und deren „Streams" priorisiert werden. Die DOM-Erstellung wird ebenfalls dadurch etwas verzögert.

Gleichzeitig haben diese Verzögerungen den Parameter „Start Render" nicht verlangsamt und in beiden Anwendungen wurde der Parameter fast gleichzeitig angezeigt. Dies bedeutet, dass sich im aktuellen Fall die Übergabe von zusätzlichen Schriftdateien lohnt, weil diese deutlich schneller auf dem Viewport erscheinen werden. Dies zeigt sich auch unter mobilen Netzwerken. Allerdings muss man beachten, dass gepushte Dateien alle anderen Ressourcen auf der Seite bremsen werden. Deshalb kann man sagen, dass der „Server Push" – Einsatz für Schriften sich dann lohnt, wenn nicht zu viele Schriften per „Server Push" übergegeben werden.

Obwohl im Test bei der Anwendung von „Server Push" für CSS Dateien und Schriften im Vergleich zur Anwendung von „Server Push" nur für CSS Dateien noch mehr Anfragebytes gespart werden, wird die Zeit zum Laden aller Ressourcen verlangsamt. Im Durchschnitt liegt diese Verzögerung bei 50 ms bei Kabel-Verbindung und bei fast 200 ms unter 3G-Verbindung.

5.5.6. Test 4: „Server Push“ – Einsatz für kritische CSS Ressourcen und nicht kritische Ressourcen (Bilder)

Bilder sind genauso wie Schriften für die Erstdarstellung auf der Seite nicht wichtig. Wenn diese Ressourcen per „Server Push“ übergeben werden, müssen alle anderen Ressourcen auf der Seite darauf warten, bis diese heruntergeladen werden. Allerdings könnte es auch Beispiele geben, in denen der sichtbare Teil der Internetseite fast nur aus Bildern besteht. In diesem Fall ist es interessant zu untersuchen, ob die Bilder mithilfe von „Server Push“ schneller im Viewport erscheinen können. Zusätzlich ist es interessant zu schauen, wie sehr die Gesamtladezeit verkürzt werden kann, wenn mehrere Bytes für Requests gespart werden.

Testbedingungen: untersucht wurde die Dozenten-Seite der Webapplikation. Im Fall der Anwendung von „Server Push“ für CSS Dateien und Bilder, wurden alle für die Seite benötigten CSS Dateien (acht Stück) und 43 kleine Bilder per „Server Push“ übergeben. Im Fall der Anwendung von „Server Push“ nur für CSS Dateien wurden acht CSS Dateien per „Server Push“ übergeben.

Im aktuellen Test wird geprüft, wie weit die wichtigsten Parameter des kritischen Rendering – Pfades durch den Einsatz von „Server Push“ für CSS Dateien und Bilder beeinflusst werden. Außerdem wird untersucht, wie sich die Ladezeit der Webapplikation und die Darstellung im Viewport ändern werden.

„Time To First Byte“

Wenn man beide Boxplot-Diagramme aus Abb. 35 miteinander vergleicht, sieht man, dass der Parameter „Time To First Byte“ für die erste Messreihe etwas später ausgegeben wird: die unteren 75% aller Werte aus der zweiten Messreihe liegen niedriger, als die ersten Werte aus der ersten Messreihe. Der Unterschied zwischen den Medianwerten liegt bei 6 ms.

Der Grund für diese Verzögerung wurde bereits bei der Funktionsweise des „Server Pushs“ beschrieben und in diesem Fall bestätigt: je mehr Dateien per „Server Push“ übergegeben werden, desto mehr wird der Parameter „Time To First Byte“ verzögert. Im Test mit der Anwendung von „Server Push“ für CSS Dateien und Bilder, werden insgesamt 51 Dateien per „Server Push“ übertragen.

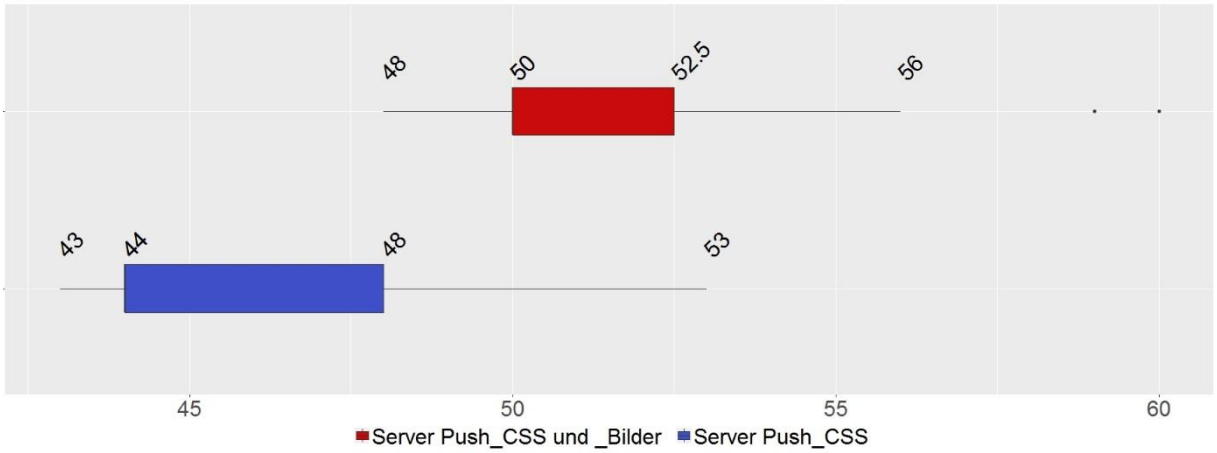

Abb. 35: Vergleich des Parameters „Time To First Byte" für die Anwendung von „Server Push" für CSS Dateien und Bilder mit der Anwendung von „Server Push" nur für CSS Dateien, jeweils mit Kabel-Verbindung.

Parameter	„Server Push" mit CSS Dateien und Bildern	„Server Push" mit CSS Dateien
Median	50 ms	44 ms
Unteres Quartil	50 ms	44 ms
Oberes Quartil	52,5 ms	48 ms

Tab. 17: Vergleich des Parameters „Time To First Byte" für die Anwendung von „Server Push" für CSS Dateien und Bilder mit der Anwendung von „Server Push" nur für CSS Dateien, jeweils mit Kabel-Verbindung.

DOM Erstellung

Wenn man beide Boxplot-Diagramme aus Abb. 36 betrachtet, sieht man, dass die Werte für die erste Messreihe später ausgegeben werden, als für die zweite Messreihe. Die Ausgabe der Werte der ersten Messreihe beginnt erst dann, wenn schon ca. 75% aller Daten der zweiten Messreihe ausgegeben sind. Der Unterschied zwischen den Medianwerten beträgt 18 ms.

Der Grund ist der Gleiche, wie in den früheren Tests: bevor die HTML Datei zu Ende geparst wird, werden alle PUSH_PROMISE – Frames den Client erreichen und die dazugehörigen „Streams" werden priorisiert. Dies bremst die Erstellung des DOMs.

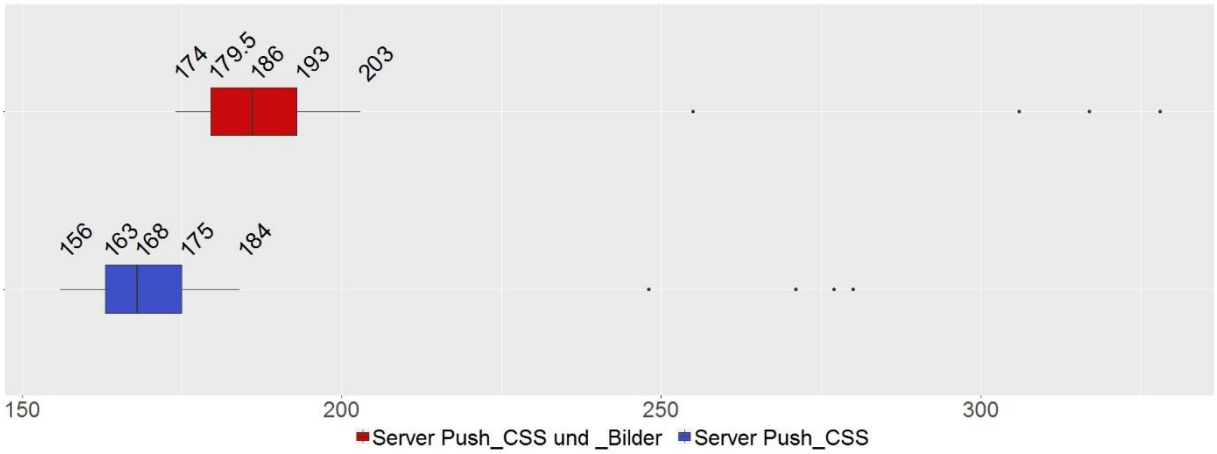

Abb. 36: Vergleich der benötigten Zeiten für die Erstellung des DOMs für 1. die Anwendung von „Server Push" für CSS Dateien und Bilder und 2. für die Anwendung von „Server Push" nur für CSS Dateien, jeweils mit Kabel-Verbindung.

Parameter	„Server Push" mit CSS Dateien und Bildern	„Server Push" mit CSS Dateien
Median	186 ms	168 ms
Unteres Quartil	179,5 ms	163 ms
Oberes Quartil	193 ms	175 ms

Tab. 18: Vergleich der benötigten Zeiten für die Erstellung des DOMs für 1. die Anwendung von „Server Push" für CSS Dateien und Bilder und 2. für die Anwendung von „Server Push" nur für CSS Dateien, jeweils mit Kabel-Verbindung.

„Start Render"

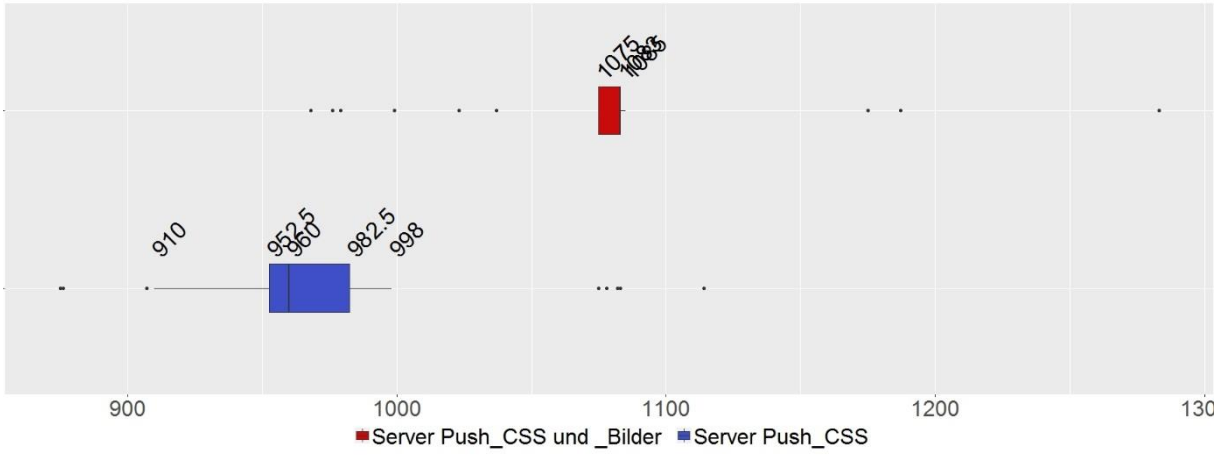

Abb. 37: Vergleich des Parameters „Start Render" für die Anwendung von „Server Push" für CSS Dateien und Bilder mit der Anwendung von „Server Push" nur für CSS Dateien, jeweils mit Kabel-Verbindung.

Parameter	„Server Push“ für CSS Dateien und Bildern	„Server Push“ für CSS Dateien
Median	1083 ms	960 ms
Unteres Quartil	1075 ms	952,5 ms
Oberes Quartil	1085 ms	982,5 ms

Tab. 19: Vergleich des Parameters „Start Render“ für die Anwendung von „Server Push“ für CSS Dateien und Bilder mit der Anwendung von „Server Push“ nur für CSS Dateien, jeweils mit Kabel-Verbindung.

Wenn man beide Boxplot-Diagramme aus Abb. 37 miteinander vergleicht, sieht man, dass die Werte der ersten Messreihe deutlich später ausgegeben werden. Der Unterschied zwischen den Medianwerten liegt bei 123 ms.

Bei dem Parameter „Start Render“ sind 123 ms Verzögerung schon kritisch. In beiden Fällen werden alle CSS Dateien per „Server Push“ übergeben, damit das CSSOM so schnell wie möglich gebaut wird. Die große Verzögerung des „Start Render“ in der ersten Messreihe kann an der Verzögerung der DOM-Erstellung liegen: das DOM wird nicht erstellt, bevor nicht alle PUSH_PROMISE – Frames den Client erreicht haben und alle dazugehörigen „Streams“ vom Client priorisiert wurden. Je mehr Dateien per „Server Push“ übergegeben werden, desto mehr wird also die DOM-Erstellung verzögert und desto wahrscheinlicher ist es, dass der Render – Baum später gebaut wird.

„Bytes Out“

Mithilfe von „Server Push“ für CSS Dateien und Bilder ist es gelungen, im Vergleich zu den Testreihen mit „Server Push nur für CSS Dateien 1490 Bytes pro Testreihe zu sparen.

„domComplete“

Aus Abb. 38 ist zu erkennen, dass alle Werte für die erste Messreihe schon fertig ausgegeben sind, bevor der untere Whisker für den zweiten Test angezeigt wird. Der Unterschied zwischen den Medianwerten beträgt 52 ms. Diese Verhaltensweise liegt vermutlich daran, dass im aktuellen Fall mehrere Bytes für Requests gespart werden. Dadurch wird die Gesamtladezeit für alle Ressourcen verkürzt.

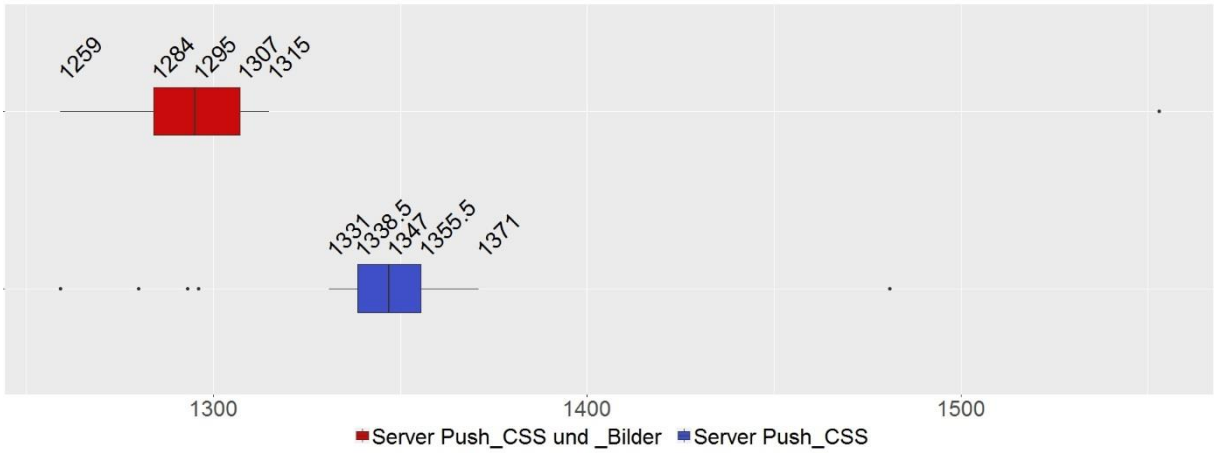

Abb. 38: Vergleich des Parameters „domComplete" für die Anwendung von „Server Push" für CSS Dateien und Bildern mit der Anwendung von „Server Push" nur für CSS Dateien, jeweils mit Kabel-Verbindung.

Parameter	„Server Push" mit CSS Dateien und Bildern	„Server Push" mit CSS Dateien
Median	1295 ms	1347 ms
Unteres Quartil	1284 ms	1338,5 ms
Oberes Quartil	1307 ms	1355,5 ms

Tab. 20: Vergleich des Parameters „domComplete" für die Anwendung von „Server Push" für CSS Dateien und Bildern mit der Anwendung von „Server Push" nur für CSS Dateien, jeweils mit Kabel-Verbindung.

„Visual Progress"

Aus Abb. 39 sieht man leider nur den „Visual Progress" der Anwendung von „Server Push" nur für CSS Dateien, weil die Werte dieser Anwendung die Werte der anderen Anwendung überlappen. Man sieht, dass beide Anwendungen gleichmäßig die Inhalte an den Client liefern. In zwei von drei Tests erscheinen bei der Anwendung von „Server Push" nur für CSS Dateien erste Inhalte im Browser früher. Die gleichen Ergebnisse sieht man aus den aufgenommenen Screenshots.

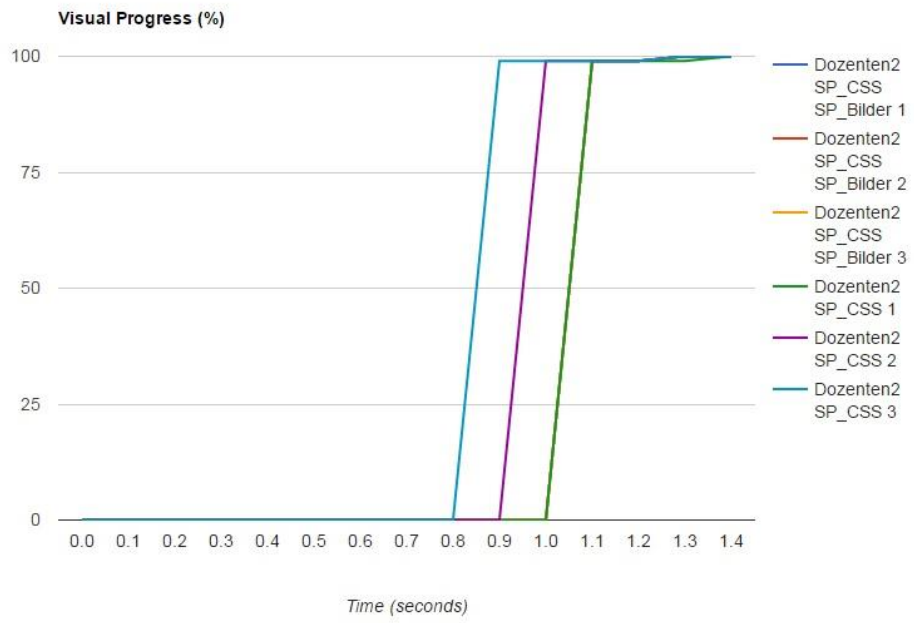

Abb. 39: „Visual Progress" bei Anwendung von „Server Push" für CSS Dateien und Bilder und bei der Anwendung von „Server Push" nur für CSS Dateien unter Kabel-Verbindung.

5.5.7. Zwischenfazit: lohnt sich die Übergabe von CSS Dateien und Bildern per „Server Push" im Vergleich zur Anwendung von „Server Push" nur für CSS Dateien?

Wenn man den Parameter „Time To First Byte" und die Zeit für die DOM-Erstellung in beiden Anwendungen vergleicht, sieht man, dass diese im Fall der Anwendung von „Server Push" für CSS Dateien und Bilder deutlich später ausgegeben werden. Dadurch, dass das DOM später aufgebaut wird, wird auch der Render – Baum später fertig gestellt und die Erstdarstellung im Viewport wird dementsprechend verzögert.

Deshalb kann gesagt werden, dass nicht eine unendliche Anzahl von Ressourcen per „Server Push" übergeben werden sollte. Auch wenn die Erstellung des CSSOMs nicht gebremst wird, kann es passieren, dass durch viele PUSH_PROMISE – Frames und die Priorisierungen der dazugehörigen „Streams" der DOM-Aufbau länger brauchen wird.

Im aktuellen Test wurden bei der Anwendung von „Server Push" für CSS Dateien und Bilder etwa 1490 Anfragebytes pro Testreihe gespart. Deshalb wird weniger Zeit für die Gesamtladezeit gebraucht. Die Ressourcen werden dadurch schneller geladen.

5.6. Zwischenfazit: „Server Push" Anwendung allgemein

Welche Ressourcentypen sollen gepusht werden? Wie groß sind die Vorteile davon?

Nachdem der Test mit per „Server Push" übergebenen CSS Dateien gute Ergebnisse gezeigt hat, wird empfohlen, alle für die zu ladende Seite relevanten CSS Dateien per „Server Push" zu übergeben. Dadurch werden mehrere Anfragebytes gespart und gepushte Ressourcen werden

den Client schneller erreichen. Da DOM und CSSOM unabhängig voneinander gebaut werden (Grigorik 2013, 168), wird CSSOM im Vergleich zur Anwendung ohne „Server Push" früher aufgebaut. Deshalb wird der Parameter „Start Render" bei der Anwendung von „Server Push" mit CSS Dateien schneller angezeigt. Dies kann eine große Rolle bei der User Experience spielen.

Außerdem wird normalerweise mit zunehmender Anzahl von gepushten Ressourcen die Gesamtladezeit verkürzt, weil Anfragebytes gespart werden.

Der „Server Push" – Einsatz kann auch für die Erstdarstellung nicht-kritischer Dateien Vorteile bringen. Falls sich auf der Seite der Webapplikation bedeutsame Schriften befinden, die gleich auf dem Viewport angezeigt werden sollen, können diese auch per „Server Push" übergegeben werden. Dadurch werden diese gleichzeitig oder kurz nach dem Erscheinen von ersten Pixeln angezeigt. Das gleiche gilt für z.B. ein Logo oder ein bedeutsames Bild auf der Seite. Wenn auch alle für die Erstdarstellung kritische Dateien per „Server Push" mitübergegeben wurden, können alle Typen (nach MIME-Typ) von Ressourcen mitgepusht werden.

Allerdings muss man beachten, dass mit jeder gepushten Ressource die DOM-Erstellung verzögert wird und alle anderen Ressourcen auf der Seite auf den Moment warten werden, bis alle gepushten Ressourcen heruntergeladen wurden.

Welche Ressourcentypen sollen nicht gepusht werden? Wo bestehen die Nachteile?

Nach den durchgeführten Tests kann man sagen, dass fast alle für die Erstdarstellung kritische und nicht-kritische Ressourcentypen gepusht werden können. Dies betrifft auch Bilder und Schriften. Lediglich nicht kritische JavaScript Dateien sind eine Ausnahme. Attribute wie „async" oder „defer" sind nur Hinweise für den Browser, die der Server nicht versteht. Aus diesem Grund werden alle JavaScript Dateien, die gepusht werden, nach der HTML Datei geladen und gleich ausgeführt. Wenn diese nicht-kritisch sind, werden erste Bytes im Viewport erst viel später angezeigt, weil JavaScript Dateien gleich ausgeführt werden (Grigorik 2016f).

Alle oben beschriebenen Ergebnisse beziehen sich auf eine konkrete Version der serverseitigen Implementation des HTTP/2 – Protokolls und eine konkrete Version des Browsers. Unter anderen Bedingungen können sich oben beschriebene Ergebnisse ändern.

5.7. Vergleich des HTTP/1.1 mit dem HTTP/2 – Protokoll

In diesem Unterkapitel werden mehrere Tests durchgeführt, in denen untersucht wird, welches der Protokolle, HTTP/1.1 oder HTTP/2, bessere Ergebnisse für den kritischen Rendering – Pfad liefert. Um das HTTP/2 – Protokoll benutzen zu können, muss eine verschlüsselte Verbindung aufgebaut werden (Browser akzeptieren das HTTP/2 – Protokoll nur unter verschlüsselten Verbindungen (NGINX, Inc. 2015, 3)). Deshalb wird im HTTP/2 – Protokoll zusätzlich zur TCP - Verbindung zum Server noch eine TLS – Verbindung aufgebaut. Es muss allerdings nur eine Verbindung pro Domain aufgebaut werden.

Wenn die Webapplikation mithilfe des HTTP/1.1 – Protokolls unter einer verschlüsselten Verbindung untersucht wird, werden alle Parameter des kritischen Rendering – Pfades verzögert. Der Grund liegt an den zahlreichen TCP – Verbindungen zwischen Client und Server. Wenn die Verbindung verschlüsselt sein soll, werden alle TCP – Verbindungen (mind. 6 TCP – Verbindungen) eine zusätzliche TLS – Verschlüsselung haben. Um diese Verbindungen herzustellen, brauchen Client und Server mehr Zeit.

Um diese Unterschiede zu zeigen, wurde noch ein kleiner Test gemacht. Da in jeder Testreihe die Zeit für den Verbindungsaufbau zum Server ungefähr gleich ist, wurden von neun Tests die Medianwerte für DNS-, TCP- und TLS- Verbindungen unter Kabel- und 3G- Verbindungen ermittelt und verglichen. Verglichen wurde die Zeit zum Verbindungsaufbau unter unverschlüsseltem HTTP/1.1 – Protokoll mit der Zeit für HTTP/1.1 mit TLS - Verschlüsselung. Untersucht wurde die Startseite der Webapplikation. Alle Ressourcen wurden für das HTTP/1.1 – Protokoll aufgeteilt.

TCP, DNS, TLS

	DNS	TCP	TLS
HTTP/1.1, Unverschlüsselt	30	228	0
HTTP/1.1, mit TLS - Verschlüsselung	61	296	353
Summe der Netzwerkverbindungszeit			
HTTP/1.1, Unverschlüsselt		HTTP/1.1, mit TLS - Verschlüsselung	
258		710	

Tab. 21: Vergleich von HTTP/1.1 unverschlüsselt und HTTP/1.1 verschlüsselt für die Startseite unter Kabel-Verbindung

TCP, DNS, TLS

	DNS	TCP	TLS
HTTP/1.1, Unverschlüsselt	310	2511	0
HTTP/1.1, mit TLS - Verschlüsselung	618	3793	4529
Summe der Netzwerkverbindungszeit			
HTTP/1.1, Unverschlüsselt		HTTP/1.1, mit TLS - Verschlüsselung	
2821		8940	

Tab. 22: Vergleich von HTTP/1.1 unverschlüsselt und HTTP/1.1 verschlüsselt für die Startseite unter 3G-Verbindung

Aus Tabelle 21 und Tabelle 22 ist zu sehen, dass die Unterschiede im Verbindungsaufbau zwischen dem verschlüsselten und dem unverschlüsselten HTTP/1.1 – Protokoll sehr groß sind. Je schlechter die Netzwerkgeschwindigkeit und je höher die Latenz ist, desto größer werden diese Unterschiede. Die lange Netzwerkverbindungszeit für HTTP/1.1 mit TLS - Verschlüsselung unter

3G-Verbindung liegt daran, dass bis zu 10 TCP – Verbindungen innerhalb einer Sitzung aufgebaut werden. Aus diesem Grund wird es zusätzliche Schwierigkeiten geben, um die Parameter des kritischen Rendering – Pfades unter verschlüsselten Verbindungen innerhalb des HTTP/1.1 – Protokolls zu untersuchen. Deshalb werden Parameterunterschiede zwischen dem HTTP/1.1- (ohne TLS - Verschlüsselung) und HTTP/2 – Protokoll (mit TLS - Verschlüsselung) gemessen.

In den Tests dieses Unterkapitels wird der „Server Push" – Einsatz für das HTTP/2 – Protokoll nicht verwendet. Deshalb wird der Parameter „Time To First Byte", „Bytes Out" und die Zeit für die DOM-Erstellung nicht mehr gemessen. Interessant zu vergleichen sind folgende Parameter:

- Verbindungsaufbau: DNS, TCP, TLS.
- „Start Render".
  An welchem Zeitpunkt fängt der Webbrowser an, etwas im Viewport zu zeichnen?
- „domComplete".
- „Visual Progress".
  Um zu schauen, wie schnell die Inhalte der Webapplikation (prozentual gesehen) auf dem Viewport erscheinen.

In diesem Unterkapitel werden zwei Tests durchgeführt, einmal für die Startseite und einmal für die Dozenten-Seite, jeweils unter Kabel-Verbindung und unter 3G-Verbindung. Die Webapplikation wird separat sowohl für das HTTP/1.1 – Protokoll als auch für das HTTP/2 – Protokoll optimiert. Für das HTTP/1.1 – Protokoll werden die Ressourcen für die Startseite zusammengefasst, sodass fünf CSS Dateien und fünf JavaScript Dateien entstanden sind. Für das HTTP/2 – Protokoll werden die gleichen Ressourcen auf mehrere Dateien aufgeteilt, so dass 10 CSS Dateien und 15 JavaScript Dateien entstehen. Das gleiche wurde für die Dozenten-Seite gemacht. Darüber hinaus werden alle Bilder für das HTTP/1.1 – Protokoll in einem Sprite zusammengefasst und unter dem HTTP/2 – Protokoll wird dieses auf einzelne kleine Bilder aufgeteilt.

Während der Testauswertung ist ein großer Unterschied zwischen der Verarbeitungsart der Ressourcen unter dem HTTP/1.1- und dem HTTP/2 – Protokoll aufgefallen. Dies wurde bei den Testergebnissen aus dem „Webpagetest.org" – Tool registriert. Während der Aufbau der Webapplikation für beide Protokolle ähnlich ist (außer, dass die Ressourcen anders aufgeteilt sind), unterscheidet sich die Reihenfolge der geladenen Ressourcen.

Der Aufbau der Webapplikation in den zu untersuchenden Tests ist immer der folgende: zuerst befinden sich im Head-Bereich des HTML-Dokuments alle CSS Dateien, danach befinden sich im Body-Bereich der gesamte Content mit allen Bildern und bevor sich der Body-Tag schließt, stehen alle JavaScript Dateien. Aus dem Ressourcenwasserfall im „Webpagetest.org" – Tool sieht man, dass die Ressourcen unter dem HTTP/1 – Protokoll in folgender Reihenfolge geladen werden: HTML Datei, danach alle CSS Dateien, danach alle JavaScript Dateien und zuletzt werden die Bilder und Schriften geladen. Außerdem muss man beachten, dass alle Ressourcen

auf einzelne TCP – Verbindungen aufgeteilt sind. Innerhalb jeder TCP – Verbindung kann man oben beschriebene Reihenfolge beobachten. Dies kann man mithilfe des „Webpagetest.org" – Tools in „Connect View" beobachten.

Wenn die Webapplikation unter dem HTTP/2 – Protokoll läuft, ändert sich die Reihenfolge der geladenen Dateien. Aus dem Ressourcenwasserfall und dem „Connect View" des „Webpagetest.org" – Tools sieht man, dass alle Ressourcen in der folgenden Reihenfolge geladen werden: HTML Datei, danach alle CSS Dateien, danach werden alle Bilder und zu Letzt die JavaScript Dateien und Schriften geladen. Diese Situation sieht etwas anders aus, wenn die Bilder durch CSS Dateien geladen werden. Dann werden alle JavaScript Dateien vor den Bildern aus der CSS Datei geladen.

Diese Unterschiede in der Reihenfolge der geladenen Ressourcen unter dem HTTP/1.1- und dem HTTP/2 – Protokoll lassen sich nicht einfach erklären. Deshalb wurde die Webapplikation mit der gleichen Browserversion wie im „Webpagetest.org" – Tool am lokalen PC aufgerufen und mithilfe von „Wireshark" zusätzlich untersucht.

Wenn man den „Wireshark" Datenmitschnitt auf dem lokalen PC betrachtet, sieht man, dass die Ressourcen unter dem HTTP/2 – Protokoll eine andere Ladereihenfolge haben: zuerst wird die HTML Datei aufgerufen, danach werden die CSS Dateien geladen, schließlich die JavaScript Dateien heruntergeladen und am Ende kommen Bilder und Schriften.

Es ist zu beachten, dass die Ladereihenfolge der Ressourcen sich je nach Browser und Browserversion unterscheiden kann. Das genaue Verhalten der Ladereihenfolge von Ressourcen ist nicht dokumentiert und nicht spezifiziert (Mifsud 2016).

Testbedingungen für das HTTP/1.1- und das HTTP/2 – Protokoll

Nachdem die oben beschriebenen Unterschiede bemerkt wurden, wurden die Tests für jedes Protokoll noch zusätzlich angepasst. Die Anpassungen betreffen die JavaScript Dateien. Unter dem HTTP/1.1 – Protokoll werden alle JavaScript Dateien erst nach den CSS Dateien vom Client angefragt. Um die DOM-Erstellung nicht zu verzögern kann man entweder das Attribut „async" oder „defer" benutzen. Wenn man das Attribut „defer" verwendet, besteht die Gefahr, dass der gesamte Content (inkl. aller Bilder) darauf warten wird, bis alle Skripte ausgeführt werden. Dies passiert, da die mit dem Attribut „defer" markierten Skripte nach dem Parameter „domInteractive" (gleich nach dem DOM-Aufbau) ausgeführt werden. Für weniger wichtige JavaScript Dateien ist es keine gute Lösung unter dem HTTP/1.1 - Protokoll.

Wenn jedoch das Attribut „async" für alle JavaScript Dateien angewendet wird, besteht die Gefahr, dass Skripte in unterschiedlicher Reihenfolge (soweit sie geladen werden) ausgeführt werden. Dann ist zu erwarten, dass früher geladene kleinere JavaScript Dateien früher ausgeführt werden, als größere Dateien. Dadurch kann es zu JavaScript-Fehlern kommen, da die Skripte oftmals voneinander abhängig sind.

Um für das HTTP/1.1 – Protokoll ein Balance zu finden, wurde entschieden, das Attribut „async" für alle von anderen abhängigen Skripte zu verwenden. Die sofort notwendigen JavaScript Dateien wurden ohne Attribute leer gelassen. In den aktuellen Tests betrifft dies nur zwei JavaScript Dateien: „jquery.min.js" und „jssor.slider.mini.js". Alle anderen werden mit dem Attribut „async" markiert.

Für den kritischen Rendering - Pfad ist es immer wichtig, die DOM- und CSSOM-Erstellung nicht zu verzögern. Wie unter dem HTTP/2 – Protokoll zuvor bemerkt wurde, werden die Skripte erstaunlicherweise erst nach den Bildern geladen. Wenn man für manche Skripte, wie unter dem HTTP/1.1 – Protokoll das Attribut „async" verwendet, wird die DOM-Erstellung wegen der nicht mit dem Attribut „async" markierten Skripte verzögert. Deshalb wurde entschieden, unter dem HTTP/2 – Protokoll das Attribut „defer" für alle Skripte anzuwenden.

5.7.1. Test 1: Untersuchung der Startseite der Webapplikation unter dem HTTP/1.1- und dem HTTP/2 – Protokoll.

Da in jedem Test die Zeit für den Verbindungsaufbau zum Server ungefähr gleich ist, wurden für neun Tests die Medianwerte für DNS-, TCP- und TLS- Verbindungen ermittelt und unter Kabel- und 3G- Verbindungen verglichen. Verglichen wurde die Zeit zum Verbindungsaufbau unter dem HTTP/1.1 – Protokoll ohne Verschlüsselung und unter dem HTTP/2 – Protokoll.

TCP, DNS, TLS

	DNS	TCP	TLS
HTTP/2	59	84	98
HTTP/1.1	30	228	0
Summe der Netzwerkverbindungszeit			
HTTP/2		HTTP/1.1	
241		258	

Tab. 23: Vergleich von HTTP/1.1 unverschlüsselt und HTTP/2 – verschlüsselt für die Startseite unter Kabel-Verbindung

TCP, DNS, TLS

	DNS	TCP	TLS
HTTP/2	621	957	654
HTTP/1.1	310	2511	0
Summe der Netzwerkverbindungszeit			
HTTP/2		HTTP/1.1	
2232		2821	

Tab. 24: Vergleich von HTTP/1.1 unverschlüsselt und HTTP/2 – verschlüsselt für die Startseite unter 3G-Verbindung

Aus Tab. 23 und Tab. 24 ist zu erkennen, dass sich die Zeit für den Verbindungsaufbau zum Server zwischen dem HTTP/1.1- und dem HTTP/2 – Protokoll wesentlich nur im Fall der 3G-Verbindung unterscheidet. Der Unterschied liegt bei etwa 600 ms. Man darf nicht vergessen, dass unter dem HTTP/1.1 – Protokoll mind. sechs TCP – Verbindungen geöffnet werden. Deshalb ist zu erwarten, dass die Anwendung unter dem HTTP/1.1 – Protokoll mit 3G-Verbindung langsamer als unter dem HTTP/2 – Protokoll geladen wird.

„Start Render"

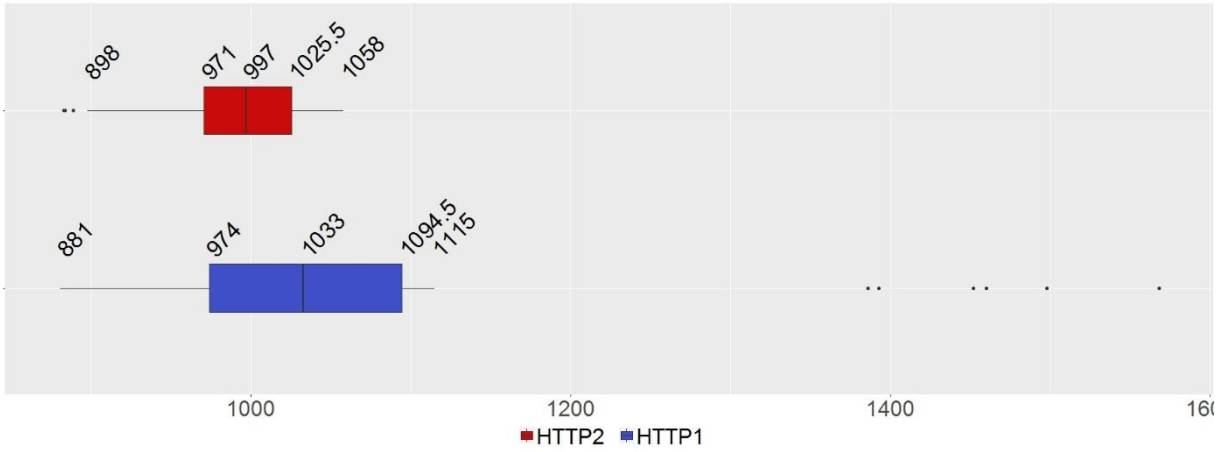

Abb. 40: Vergleich des Parameters „Start Render" unter dem HTTP/1.1- und dem HTTP/2 – Protokoll, jeweils unter Kabel-Verbindung.

Parameter	HTTP/2 – Protokoll	HTTP/1.1 – Protokoll
Median	997 ms	1033 ms
Unteres Quartil	971 ms	974 ms
Oberes Quartil	1025,5 ms	1094,5 ms

Tab. 25: Vergleich des Parameters „Start Render" unter dem HTTP/1.1- und dem HTTP/2 – Protokoll, jeweils unter Kabel-Verbindung.

Aus Abb. 40 geht hervor, dass der Parameter „Start Render" bei der Anwendung des HTTP/2 – Protokolls unter Kabel-Verbindung etwas früher als bei der Anwendung des HTTP/1.1 – Protokolls ausgegeben wird: der Median der ersten Messreihe wird 36 ms früher ausgegeben. Die ersten 75% der Werte der ersten Messreihe werden ausgegeben, bevor 50% der Werte der zweiten Messreihe angezeigt werden. Das Maximum der ersten Messreihe wird auch früher angezeigt. Dies bedeutet, dass bei der Anwendung des HTTP/2 – Protokolls erste Pixel auf dem Viewport ein wenig früher erscheinen.

Aus Abb. 41 ist zu sehen, dass der Parameter „Start Render" bei Anwendung des HTTP/2 – Protokolls unter 3G-Verbindung im Vergleich zur Anwendung des HTTP/1.1 – Protokolls deutlich später ausgegeben wird. Der Unterschied zwischen den Medianwerten beträgt 833 ms.

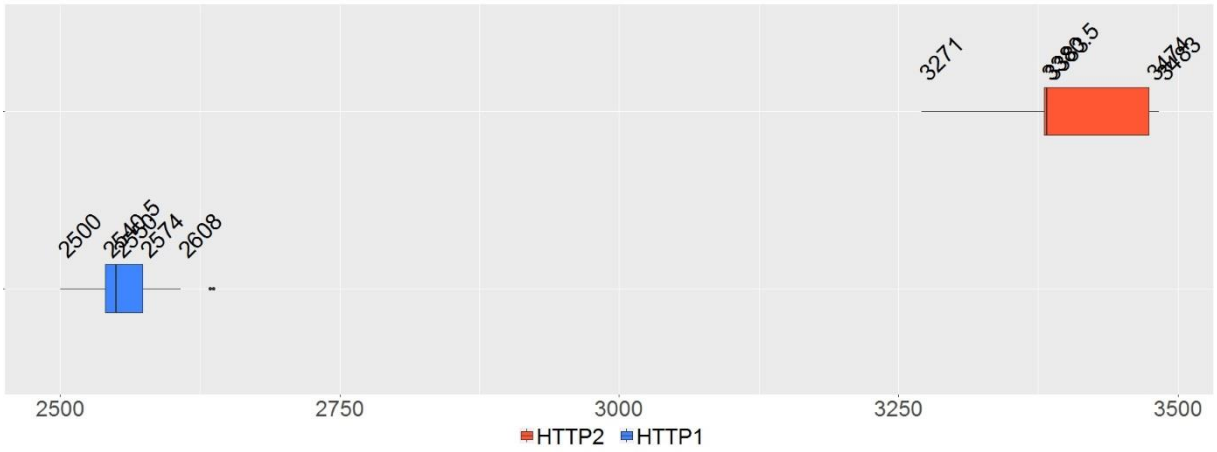

Abb. 41: Vergleich des Parameters „Start Render" unter dem HTTP/1.1- und dem HTTP/2 – Protokoll, jeweils unter 3G-Verbindung.

Parameter	HTTP/2 – Protokoll	HTTP/1.1 – Protokoll
Median	3383 ms	2550 ms
Unteres Quartil	3380,5 ms	2540 ms
Oberes Quartil	3474 ms	2574 ms

Tab. 26: Vergleich des Parameters „Start Render" unter dem HTTP/1.1- und dem HTTP/2 – Protokoll, jeweils unter 3G-Verbindung.

Wenn man die Ergebnisse unter Kabel-Verbindung mit denen unter 3G-Verbindung vergleicht, sieht man, dass der Unterschied zwischen den Medianwerten einen sehr großen Sprung auf 833 ms gemacht hat. Es gibt jedoch mehrere Faktoren, die gegen das aktuelle Testergebnis unter 3G-Verbindung sprechen:

- Aufbau von sechs (oder mehr) unverschlüsselten TCP - Verbindungen (für das HTTP/1.1 - Protokoll) unter mobilen Netzwerken braucht etwa 600 ms mehr Zeit als für eine verschlüsselte TCP – Verbindung (für das HTTP/2 – Protokoll) (siehe Vergleiche zur TCP, DNS, TLS unter 3G-Verbindung).
- Besonders unter schlechteren Netzwerk-Verbindungen wurde mehr Zeit gebraucht, um größere Dateien herunterzuladen. Dies liegt an der Funktion des TCP – Protokolls „Slow – Start" (siehe Unterkapitel „3.1 HTTP/1.1 – Optimierungstechniken, Stand bisher").
- Das HTTP/2 – Protokoll verwendet den „Header Kompression" Mechanismus, um die Kopfzeilen während der Übertragung zu komprimieren.

Unter Kabel-Verbindung wurden die Werte mit der Anwendung unter dem HTTP/2 – Protokoll sogar früher ausgegeben. Um die Antwort auf die Frage zu finden, aus welchem Grund die Webapplikation unter dem HTTP/2 – Protokoll und einer 3G-Verbindung durchschnittlich mehr als 800 ms später angezeigt wird, wird auf die genaue Funktion der aktuellen Implementierung

des HTTP/2 – Protokolls im Zwischenfazit geschaut. Da im aktuellen Fall der Parameter „Start Render" betrachtet wurde, spielt scheinbar die Aufteilung der CSS Dateien eine große Rolle.

„domComplete"

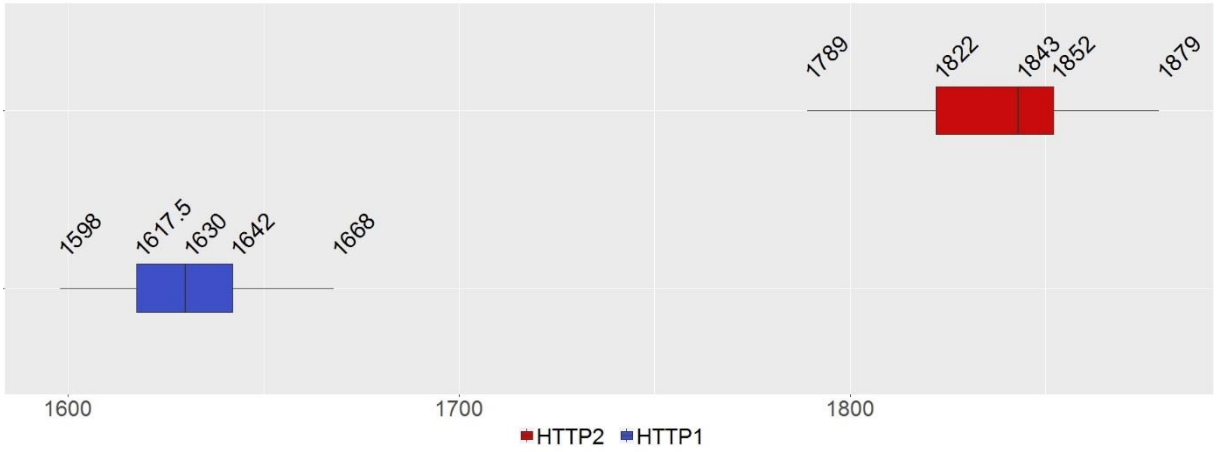

Abb. 42: Vergleich des Parameters „domComplete" bei Anwendung des HTTP/1.1- und des HTTP/2 – Protokolls, jeweils unter Kabel-Verbindung.

Parameter	HTTP/2 – Protokoll	HTTP/1.1 – Protokoll
Median	1843 ms	1630 ms
Unteres Quartil	1822 ms	1617,5 ms
Oberes Quartil	1852 ms	1642 ms

Tab. 27: Vergleich des Parameters „domComplete" bei Anwendung des HTTP/1.1- und des HTTP/2 – Protokolls, jeweils unter Kabel-Verbindung.

Aus Abb. 42 sieht man, dass der Parameter „domComplete" bei der Anwendung des HTTP/2 – Protokolls deutlich später ausgegeben wird. Der Unterschied zwischen den Median-werten liegt bei 213 ms.

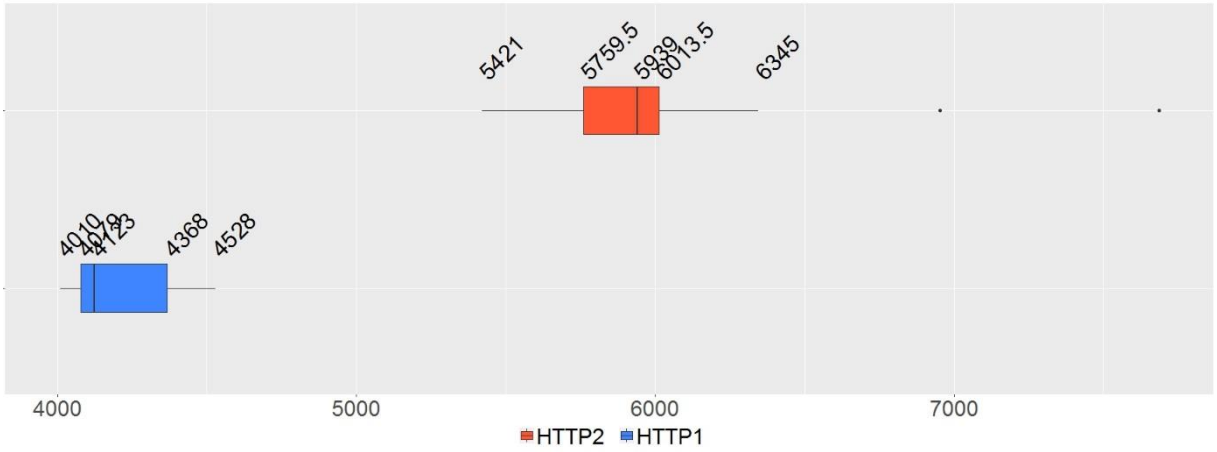

Abb. 43: Vergleich des Parameters „domComplete" bei Anwendung des HTTP/1.1- und des HTTP/2 – Protokolls, jeweils unter 3G-Verbindung.

Parameter	HTTP/2 – Protokoll	HTTP/1.1 – Protokoll
Median	5939 ms	4123 ms
Unteres Quartil	5759 ms	4079 ms
Oberes Quartil	6013 ms	4368 ms

Tab. 28: Vergleich des Parameters „domComplete" bei Anwendung des HTTP/1.1- und des HTTP/2 – Protokolls, jeweils unter 3G-Verbindung.

Aus Abb. 43 ist zu sehen, dass die Werte unter dem HTTP/2 – Protokoll deutlich später ausgegeben werden, als unter dem HTTP/1.1 – Protokoll. Der Unterschied zwischen den Medianwerten liegt bei 1816 ms. Im Vergleich zur Kabel-Verbindung hat sich dieser Unterschied um mehr als das Achtfache erhöht.

In beiden Tests unter unterschiedlichen Verbindungen sieht man, dass der Parameter „domComplete" für die Anwendung unter dem HTTP/2 – Protokoll im Vergleich zur Anwendung unter dem HTTP/1.1 – Protokoll deutlich später ausgegeben wird. Es ist eher zu erwarten, dass der Parameter „domComplete" in beiden Verbindungen unter dem HTTP/2 – Protokoll viel früher als unter dem HTTP/1.1 – Protokoll ausgegeben wird. Mögliche Faktoren wurden in der Auswertung des Parameters „Start Render" aufgelistet. Die Erwartungen haben sich beim Parameter „domComplete" nicht erfüllt.

„Visual Progress"

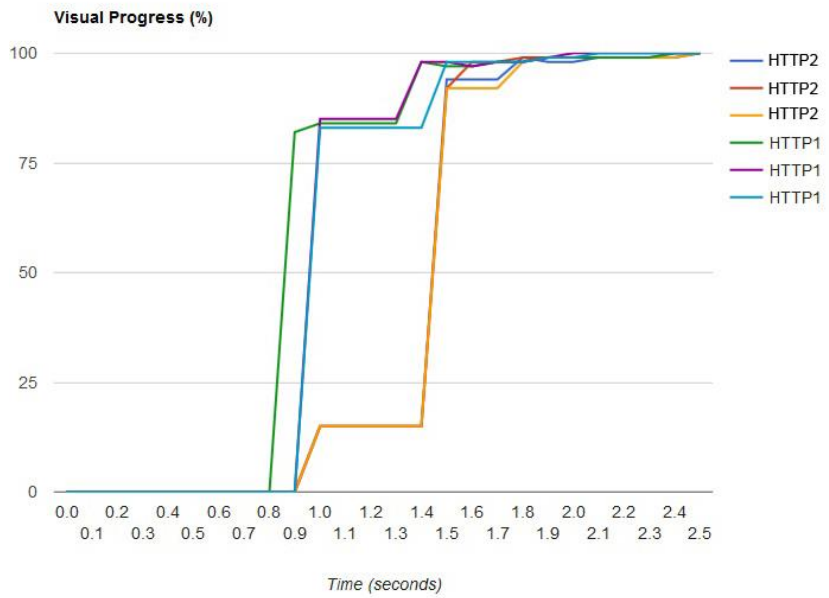

Abb. 44: „Visual Progress" bei Anwendung des HTTP/1.1- und des HTTP/2 – Protokolls unter Kabel-Verbindung.

Aus oben dargestellter Grafik sieht man, dass die Webapplikation unter beiden Protokollen etwa gleichzeitig erste Inhalte auf dem Browser anzeigt. Man sieht aber, dass die meisten In-

halte (bei etwa 80%) unter dem HTTP/1.1 – Protokoll schneller geladen werden. Dies liegt daran, dass die Dateien unter dem HTTP/2 – Protokoll eine andere Ressourcenreihenfolge haben und JavaScript Dateien mit dazugehörigen Bildern aus dem Slider später nachgefragt werden. Der Unterschied liegt bei etwa einer halben Sekunde.

Aus den Screenshots sieht man zusätzlich, dass unter dem HTTP/2 – Protokoll zuerst alle Bilder auf der Seite geladen werden, bevor der Slider mit den dazugehörigen Bildern im Browser erscheinen wird. Die Ressourcenreihenfolge wurde beim Ressourcenwasserfall bestätigt.

Aus Abb. 45 lässt sich schließen, dass erste Inhalte unter der HTTP/2 – Verbindung fast eine Sekunde später angezeigt werden. Die JavaScript Dateien werden wie im früheren Test später nachgeladen. Aus den Screenshots sieht man zusätzlich, dass der Unterschied beim fertig geladenen Slider zwischen dem HTTP/1.1- und dem HTTP/2 – Protokoll bei etwa 2,5 Sekunden liegt.

Aus den aufgenommenen Screenshots ist auch zu sehen, dass bei der Anwendung unter dem HTTP/2 – Protokoll alle Ressourcen deutlich später nachgeladen werden.

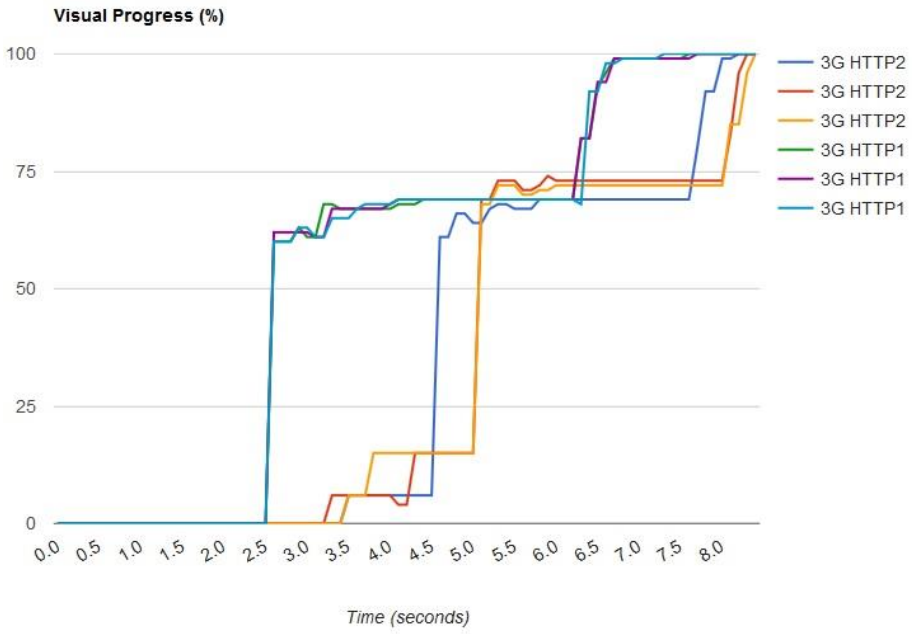

Abb. 45: „Visual Progress" bei Anwendung des HTTP/1.1- und des HTTP/2 – Protokolls, jeweils unter 3G-Verbindung.

5.7.2. Zwischenfazit: wie groß sind die Unterschiede der untersuchten Parameter des kritischen Rendering – Pfades zwischen dem HTTP/1.1- und dem HTTP/2 – Protokoll in Bezug auf die Startseite der Webapplikation?

Aus den Ergebnissen des aktuellen Tests kann man sagen, dass die Gesamtladezeit für alle Ressourcen unter dem HTTP/2 – Protokoll im Vergleich zum HTTP/1.1 – Protokoll deutlich länger dauert. Umso größer wird dieser Unterschied, je größer die Latenz ist. Dies ist besonders

verwunderlich, weil die Zeit für den Verbindungsaufbau zum Server unter dem HTTP/2 – Protokoll unter 3G-Verbindung 600 ms weniger dauert.

Der Parameter „Start Render" wird unter dem HTTP/2 – Protokoll unter Kabel-Verbindung ein wenig früher angezeigt. Aber mit zunehmender Latenz ändern sich die Ergebnisse rasant. Unter der 3G-Verbindung wird dieser Parameter unter dem HTTP/2 – Protokoll im Durchschnitt schon 800 ms später angezeigt als unter dem HTTP/1.1 – Protokoll. Diese Unterschiede sind für das User Experience sehr kritisch (Rigor, Inc. 2016).

Warum das HTTP/2 – Protokoll schlechtere Ergebnisse als das HTTP/1.1 – Protokoll liefert, ist auf den ersten Blick nicht klar. Um die Erklärung für dieses Verhalten zu finden, muss man genauer schauen, wie die Kommunikation unter dem HTTP/2 – Protokoll zwischen dem Client und dem Server abläuft. Bevor die Schlussfolgerungen zu diesem Test gemacht werden, ist es sinnvoll nach den Ergebnissen der untersuchten Parameter für die Dozenten-Seite zu schauen. Bei der Dozenten-Seite ist die „klassische" Aufteilung von allen Ressourcen zwischen dem HTTP/1.1- und dem HTTP/2 – Protokoll noch mehr zu sehen: für das HTTP/2 – Protokoll werden nicht nur CSS und JavaScript Dateien klein gehalten, sondern auch alle Bilder werden explizit aufgeteilt. Wenn der Grund der Parameterverzögerung unter dem HTTP/2 – Protokoll auf der größeren Anzahl von Ressourcen liegt, ist zu erwarten, dass die Medianwerte der zu untersuchenden Parameter sich noch stärker unterscheiden werden.

5.7.3. Test 2: Untersuchung der Dozenten-Seite unter den Protokollen HTTP/1.1- und HTTP/2.

Da in jedem Test die Zeit für den Verbindungsaufbau zum Server ungefähr gleich ist, wurden anhand von neun Tests die Medianwerte für DNS-, TCP- und TLS- Verbindungen ermittelt und unter Kabel- und 3G- Verbindung verglichen. Verglichen wurde die Zeit zum Verbindungsaufbau unter dem HTTP/1.1 – Protokoll ohne Verschlüsselung und unter dem HTTP/2 – Protokoll.

TCP, DNS, TLS

	DNS	TCP	TLS
HTTP/2 – Protokoll	60 ms	78 ms	97 ms
HTTP/1.1 - Protokoll	30 ms	255 ms	0 ms
Summe der Netzwerkverbindungszeit			
HTTP/2 – Protokoll		HTTP/1.1 – Protokoll	
235 ms		285 ms	

Tab. 29: Vergleich von HTTP/1.1 unverschlüsselt und HTTP/2 – verschlüsselt für die Dozenten-Seite unter Kabel-Verbindung

TCP, DNS, TLS

	DNS	TCP	TLS
HTTP/2 – Protokoll	627 ms	958 ms	653 ms
HTTP/1.1 - Protokoll	309 ms	1907 ms	0 ms
Summe der Netzwerkverbindungszeit			
HTTP/2 – Protokoll		HTTP/1.1 – Protokoll	
2238		2216 ms	

Tab. 30: Vergleich von HTTP/1.1 unverschlüsselt und HTTP/2 – verschlüsselt für die Dozenten-Seite unter 3G-Verbindung

Aus Tab. 29 und Tab. 30 ist zu erkennen, dass die Zeit für den Verbindungsaufbau zum Server zwischen dem HTTP/1.1- und dem HTTP/2 – Protokoll sich sowohl unter Kabel- als auch unter 3G-Verbindung nur ein wenig unterscheidet. Der Unterschied bei der Kabel-Verbindung liegt bei 50 ms und bei 3G-Verbindung bei 22 ms. Es ist interessant zu sehen, wie sich im aktuellen Test die zu untersuchenden Parameter unterscheiden werden. Falls die Webapplikation unter dem HTTP/2 – Protokoll langsamer als unter dem HTTP/1.1 – Protokoll geladen wird, wird der Grund nicht an der Netzwerkverbindungszeit liegen.

„Start Render"

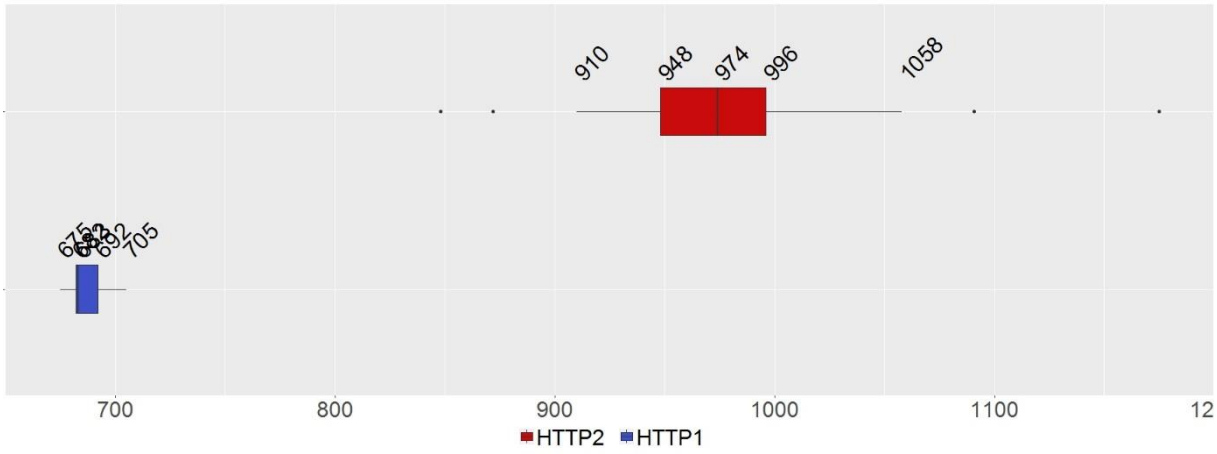

Abb. 46: Vergleich des Parameters „Start Render" bei Anwendung des HTTP/1.1- und des HTTP/2 – Protokolls unter Kabel-Verbindung.

Parameter	HTTP/2 – Protokoll	HTTP/1.1 – Protokoll
Median	974 ms	683 ms
Unteres Quartil	948 ms	682 ms
Oberes Quartil	996 ms	692 ms

Tab. 31: Vergleich des Parameters „Start Render" bei Anwendung des HTTP/1.1- und des HTTP/2 – Protokolls unter Kabel-Verbindung.

Aus der Abb. 46 ist zu sehen, dass die Parameterwerte unter dem HTTP/2 – Protokoll deutlich später als unter dem HTTP/1.1 – Protokoll ausgegeben werden. Der Unterschied zwischen den Medianwerten liegt bei 291 ms.

Wenn der Parameter „Start Render" verzögert ist, bedeutet es, dass die kritischen Dateien (im aktuellen Fall nur CSS Dateien) später zum Client geliefert werden. Diese Ergebnisse kann man nicht einfach erklären, ohne den Transport der Pakete und die Lieferreihenfolge der einzelnen Frames genauer zu untersuchen.

Aus Abb. 47 ist zu sehen, dass der Unterschied zwischen den Medianwerten bei etwa 1453 ms liegt. Für die User Experience, besonders unter mobilen Netzwerken, ist dieser Unterschied mit fast 1,5 Sekunden sehr groß (Rigor, Inc. 2016). Wenn man die Medianwerte des Tests zur Startseite mit dem aktuellen Test vergleicht, ist sofort zu sehen, dass im aktuellen Test die Unterschiede zwischen den Medianwerten des HTTP/1.1- und des HTTP/2 – Protokolls deutlich größer geworden sind.
Der Grund für diese Ergebnisse ist immer noch nicht klar. Es kann jedoch vermutet werden, dass die Verzögerung größer wird, je mehr Ressourcen vom Server angefragt werden. Im aktuellen Test werden für das HTTP/1.1 – Protokoll drei CSS Dateien vorbereitet und für das HTTP/2 – Protokoll acht CSS Dateien.

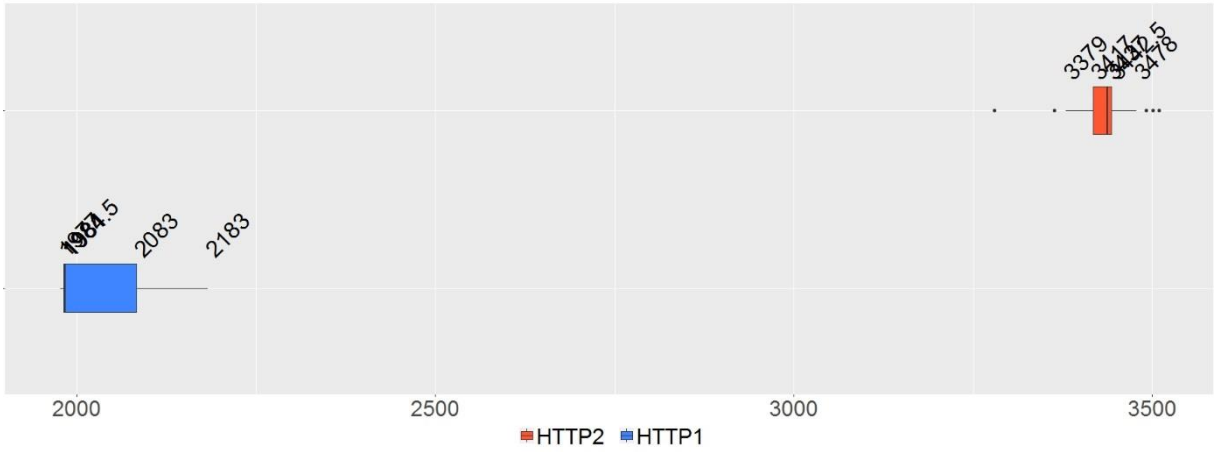

Abb. 47: Vergleich des Parameters „Start Render" bei Anwendung des HTTP/1.1- und des HTTP/2 – Protokolls unter 3G-Verbindung.

Parameter	HTTP/2 – Protokoll	HTTP/1.1 – Protokoll
Median	3437 ms	1984 ms
Unteres Quartil	3417 ms	1981,5 ms
Oberes Quartil	3442,5 ms	2083 ms

Tab. 32: Vergleich des Parameters „Start Render" bei Anwendung des HTTP/1.1- und des HTTP/2 – Protokolls unter 3G-Verbindung.

„domComplete“

Aus Abb. 48 sieht man, dass die Werte des Parameters „domComplete“ unter dem HTTP/2 – Protokoll deutlich später angezeigt werden. Der Unterschied zwischen den Medianwerten liegt bei 190 ms.

Aus Abb. 49 sieht man, dass die Werte der ersten Messreihe deutlich später ausgegeben werden. Der Unterschied zwischen den Medianwerten beträgt 2325 ms.

Man sieht immer wieder, dass der Unterschied zwischen der Gesamtladezeit des HTTP/1.1- und des HTTP/2 – Protokolls größer ist, je mehr Ressourcen unter der HTTP/2 – Verbindung übertragen werden. Man darf allerdings nicht vergessen, dass die Zeit, die für den Verbindungsaufbau zum Server benötigt wurde, unter 3G-Verbindung beim HTTP/1.1 – Protokoll im aktuellen Fall niedriger als beim HTTP/2 – Protokoll ist.

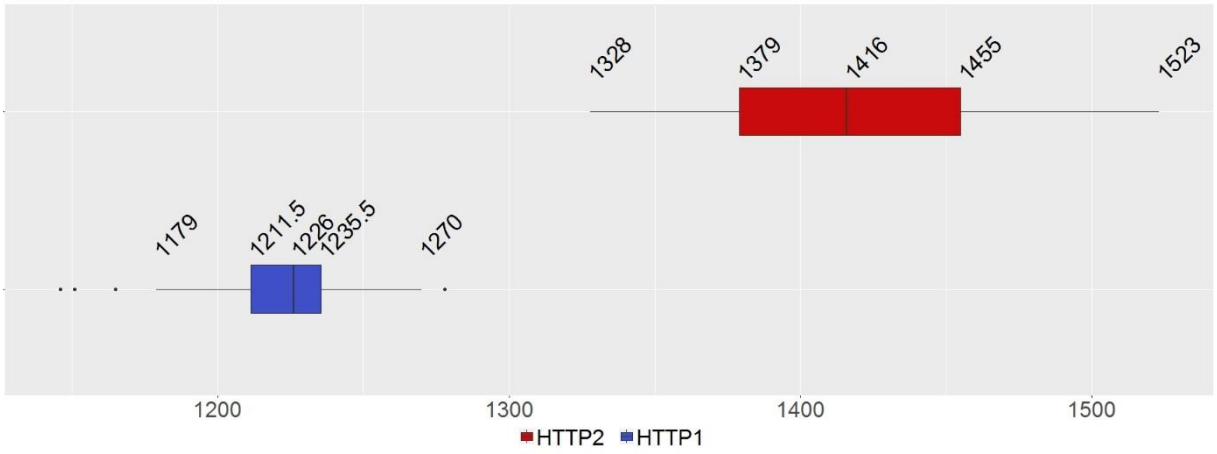

Abb. 48: Vergleich des Parameters „domComplete“ bei Anwendung des HTTP/1.1- und des HTTP/2 – Protokolls unter Kabel-Verbindung.

Parameter	HTTP/2 – Protokoll	HTTP/1.1 – Protokoll
Median	1416 ms	1226 ms
Unteres Quartil	1379 ms	1211,5 ms
Oberes Quartil	1455 ms	1235,5 ms

Tab. 33: Vergleich des Parameters „domComplete“ bei Anwendung des HTTP/1.1- und des HTTP/2 – Protokolls unter Kabel-Verbindung.

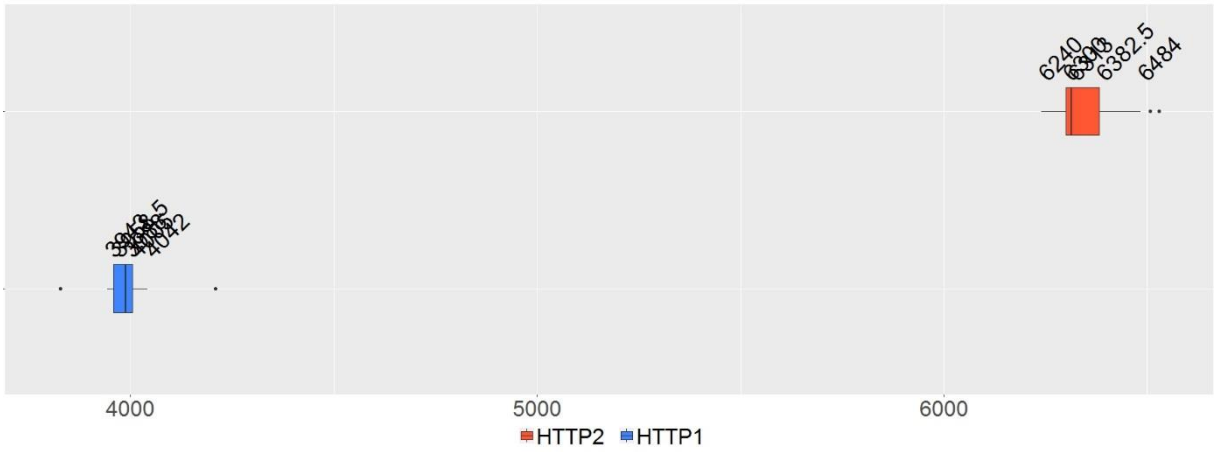

Abb. 49: Vergleich des Parameters „domComplete" bei Anwendung des HTTP/1.1- und des HTTP/2 – Protokolls unter 3G-Verbindung.

Parameter	HTTP/2 – Protokoll	HTTP/1.1 – Protokoll
Median	6313 ms	3988 ms
Unteres Quartil	6300 ms	3958,5 ms
Oberes Quartil	6382,5 ms	

Tab. 34: Vergleich des Parameters „domComplete" bei Anwendung des HTTP/1.1- und des HTTP/2 – Protokolls unter 3G-Verbindung.

„Visual Progress"

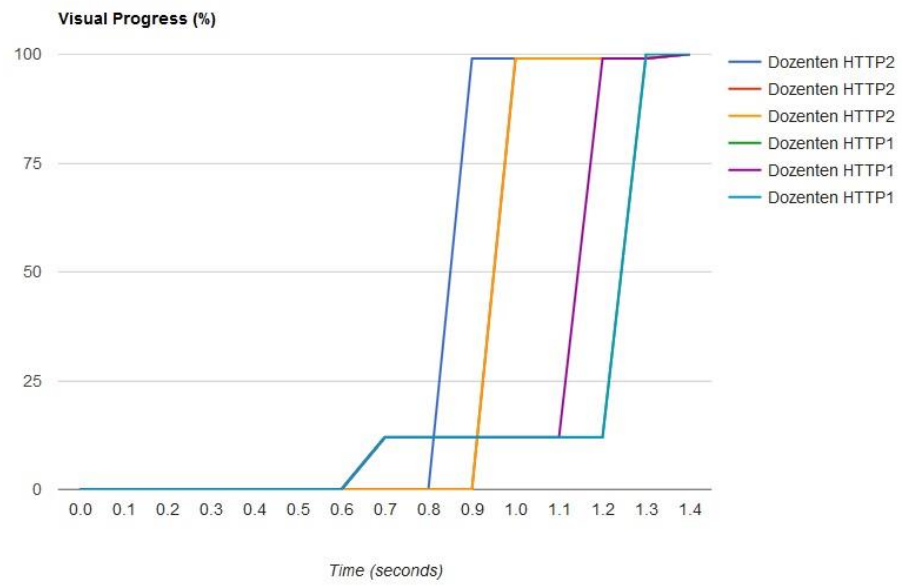

Abb. 50: „Visual Progress" bei Anwendung des HTTP/1.1- und des HTTP/2 – Protokolls auf die Dozenten-Seite unter Kabel-Verbindung.

Aus oben dargestellter Grafik sieht man, dass die ersten Inhalte der Webapplikation unter dem HTTP/1.1 – Protokoll 200 ms bis 300 ms früher angezeigt werden. Der Großteil der Inhalte wird beim HTTP/2 – Protokoll dagegen früher angezeigt.

Aus den aufgenommenen Screenshots sieht man, wie schnell die Ressourcen auf dem Viewport erscheinen. Erste Pixel mit Überschrift und Logo werden unter dem HTTP/1.1 – Protokoll etwas früher angezeigt. Unter dem HTTP/2 – Protokoll werden dagegen fast alle Ressourcen auf einmal angezeigt. D.h., dass unter dem HTTP/1 – Protokoll erste Pixel früher erscheinen, da die Ressourcen unter dem HTTP/2 – Protokoll länger zum Herunterladen brauchen, jedoch unter dem HTTP/2 – Protokoll der Hauptteil des Inhaltes früher erscheint.

Aus Abb. 51 geht hervor, dass erste Pixel der Webapplikation unter dem HTTP/2 – Protokoll etwa 1,5 Sekunden später angezeigt werden, als beim HTTP/1.1 – Protokoll. Außerdem sieht man, dass die Ressourcen beim HTTP/2 – Protokoll nacheinander „portionsweise" nachgeladen werden, während fast alle Inhalte beim HTTP/1.1 - Protokoll den Client früher erreichen.

Aus den Screenshots sieht man noch deutlicher, wie die Ressourcen unter dem HTTP/2 – Protokoll nachgeladen werden. Erste Bilder beim HTTP/2 – Protokoll werden später als das komplette Bild beim HTTP/1.1 – Protokoll angezeigt. Die Ressourcen werden beim HTTP/2 – Protokoll deutlich länger geladen.

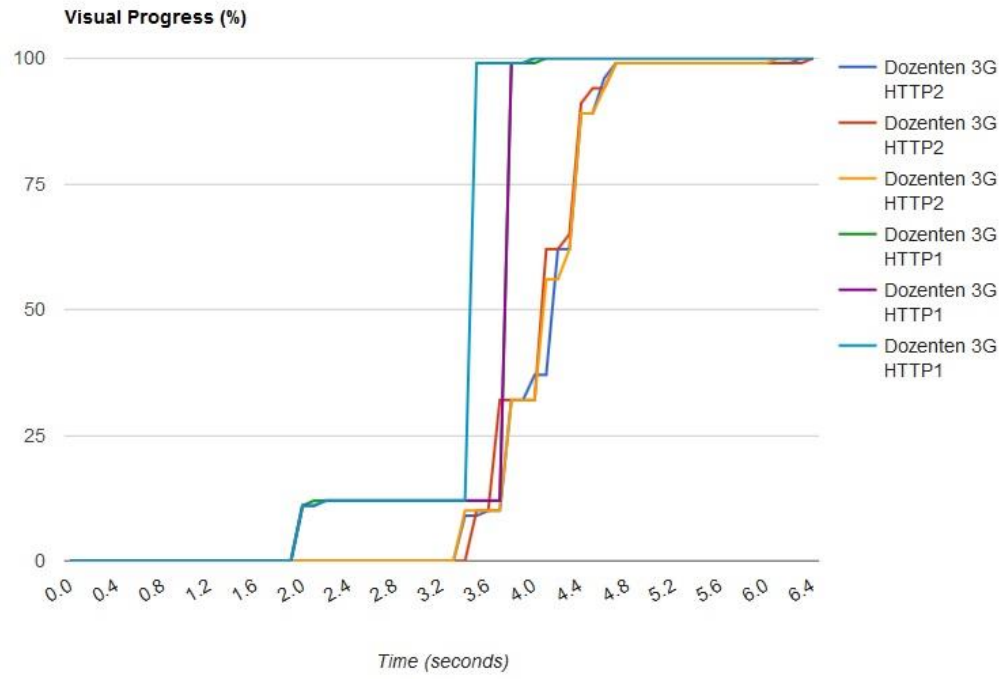

Abb. 51: „Visual Progress" bei Anwendung des HTTP/1.1- und des HTTP/2 – Protokolls auf die Dozenten-Seite unter 3G-Verbindung.

5.7.4. Zwischenfazit: wie groß sind die Unterschiede der Parameter des kritischen Rendering - Pfades zwischen dem HTTP/1.1- und dem HTTP/2 – Protokoll in Bezug auf die Dozenten-Seite der Webapplikation?

Der aktuelle Test hat große Unterschiede in den untersuchten Parametern zwischen dem HTTP/1.1- und dem HTTP/2 – Protokoll gezeigt. Wenn man die Testergebnisse des Tests der Startseite mit den Ergebnissen des aktuellen Tests vergleicht, sieht man, dass die Gesamtladezeit unter dem HTTP/2 – Protokoll größer ist, je mehr Ressourcen geladen werden. Diese Situation verstärkt sich deutlich unter der 3G-Verbindung. Gleichzeitig passiert dies unter früher genannten Bedingungen:

- Größere Dateien könnten mehr Zeit zum Herunterladen brauchen. Dies liegt an der „Slow Start" – Funktion des TCP – Protokolls (siehe Unterkapitel „3.1 HTTP/1.1 – Optimierungstechniken, Stand bisher"). Deshalb ist es sehr verwunderlich, dass kleinere Dateien unter dem HTTP/2 – Protokoll bei 3G-Verbindung viel mehr Zeit zum Herunterladen brauchen, als größere Dateien unter dem HTTP/1.1 – Protokoll.
- Das HTTP/2 – Protokoll verwendet die „Header Kompression" – Technologie, die dafür sorgt, dass nur wenige Bytes innerhalb der Sitzung transportiert werden.

Um die Antwort auf die Frage zu finden, warum die Webapplikation unter dem HTTP/2 – Protokoll deutlich langsamer lädt als unter dem HTTP/1.1 – Protokoll, sollte man den genauen Austausch der Frames zwischen Client und Server beobachten. Deshalb wird im nächsten Test untersucht, wie groß die Parameterunterschiede sind, wenn man wenige große Dateien oder mehrere kleine Dateien verwendet.

5.8. Spielt die Ressourcenaufteilung eine Rolle für das HTTP/2 – Protokoll?

Die Aufteilung der Ressourcen kann sowohl ein Performance- als auch ein Entwicklungskriterium sein. Dies gilt für jede Webapplikation: je kleiner die Ressourcen gehalten werden, desto einfacher und übersichtlicher ist es, um die Webapplikation zu entwickeln. Dies betrifft vor allem die Maßnahmen, die speziell für die Frontend Performance – Optimierung gemacht wurden, z.B. Zusammenfügen von kritischen Ressourcen und Bildern unter dem HTTP/1.1 – Protokoll. Außerdem ist es praktischer, um kleinere Dateien zu cachen, falls die Änderungen nur in einem Teil der Webapplikation stattgefunden haben. Noch ein wichtiges Kriterium ist, dass größere Dateien unter hoher Latenz, die in Mobilen Netzwerken besteht, langsamer heruntergeladen werden (siehe Unterkapitel „3.1. HTTP/1.1 – Optimierungstechniken, Stand bisher").

Wie die Theorie besagt, müssen die für das HTTP/1.1 – Protokoll unternommenen Maßnahmen zur Ressourcenzusammenfassung unter dem HTTP/2 – Protokoll nicht wiederholt werden (NGINX, Inc. 2015, 13) (siehe Unterkapitel „3.4. Mögliche Optimierungstechniken für das HTTP/2 – Protokoll"). Die Tests, in denen die Parameter des kritischen Rendering – Pfades zwischen dem HTTP/1.1- und dem HTTP/2 – Protokoll verglichen wurden, haben gezeigt, dass das HTTP/1.1 – Protokoll bessere Ergebnisse liefert. Es ist allerdings interessant zu prüfen, wie die

Übergabe von kleineren Dateien die Parameter des kritischen Rendering – Pfades unter dem HTTP/2 – Protokoll verzögern wird. Um die Unterschiede in den Parametern genauer sehen zu können, wird die Webapplikation nur unter 3G-Verbindung aufgerufen.

In diesem Test sind besonders die Parameter „Start Render", „domComplete" und „Visual Progress" interessant. Da im aktuellen Test die Anwendung von „Server Push" nicht benutzt wird, sind die Untersuchung des Parameters „Time To First Byte", „Bytes Out" und die DOM-Erstellung nicht relevant.

Testbedingungen: untersucht wird die Dozenten-Seite der Webapplikation unter dem HTTP/2 – Protokoll. Im Fall der Anwendung mit vielen kleinen Dateien werden acht CSS Dateien, 43 Bilder und 12 JavaScript Dateien verwendet. Im Fall der Anwendung mit wenig großen Dateien werden drei CSS Dateien, zwei JavaScript Dateien und ein großes Sprite-Bild benutzt. In beiden Fällen haben die Dateien die gleichen Inhalte.

„Start Render"

Aus Abb. 52 ist zu erkennen, dass der Parameter „Start Render" für die Anwendung mit vielen kleineren Dateien später ausgegeben wird. Der Unterschied zwischen den Medianwerten liegt bei 18 ms. Da sich die erste Hälfte der Box der ersten Messreihe und die zweite Hälfte der Box der zweiten Messreihe überlappen, kann man sagen, dass der Parameter „Start Render" für die Anwendung mit vielen kleineren Dateien im Durchschnitt etwas später ausgegeben wird.

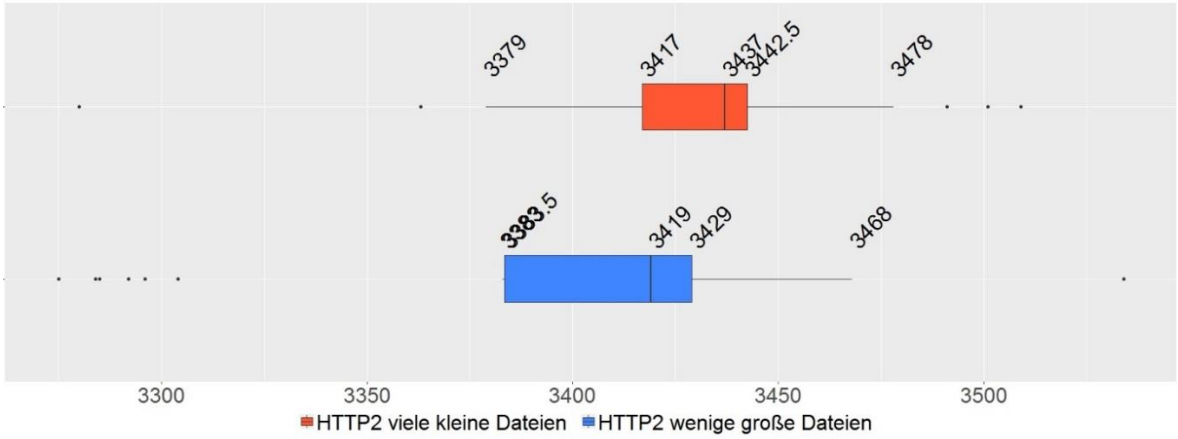

Abb. 52: Vergleich des Parameters „Start Render" bei Anwendung von vielen aufgeteilten Dateien und wenigen großen Dateien unter dem HTTP/2 – Protokoll bei 3G-Verbindung.

Parameter	HTTP/2: viele kleinere Dateien	HTTP/2: wenige größere Dateien
Median	3437 ms	3419 ms
Unteres Quartil	3417 ms	3383,5 ms
Oberes Quartil	3442,5 ms	3429 ms

Tab. 35: Vergleich des Parameters „Start Render" bei Anwendung von vielen aufgeteilten Dateien und wenigen großen Dateien unter dem HTTP/2 – Protokoll bei 3G-Verbindung.

Diese Unterschiede sind im Durchschnitt nicht so groß. Trotzdem wiedersprechen diese Ergebnisse den Erwartungen. Die Schlussfolgerungen zu diesen Ergebnissen werden im Zwischenfazit beschrieben.

„domComplete"

Aus Abb. 53 lässt sich schließen, dass die Anwendung mit vielen aufgeteilten Dateien deutlich mehr Zeit für das Herunterladen aller Ressourcen braucht. Der Unterschied zwischen den Medianwerten liegt bei 1676 ms. Diese Ergebnisse sind nicht sofort erklärbar und die Vermutungen werden im Zwischenfazit beschrieben.

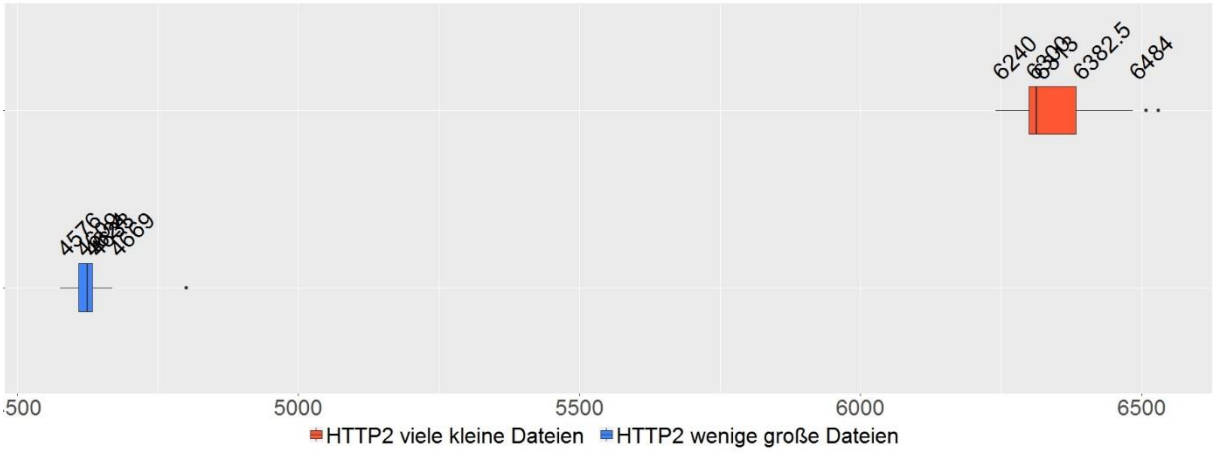

Abb. 53: Vergleich des Parameters „Start Render" bei Anwendung von vielen aufgeteilten Dateien und wenigen großen Dateien unter dem HTTP/2 – Protokoll bei 3G-Verbindung.

Parameter	HTTP/2: viele kleinere Dateien	HTTP/2: wenige größere Dateien
Median	6300 ms	4624 ms
Unteres Quartil	6382,5 ms	4609 ms
Oberes Quartil	6313 ms	4633 ms

Tab. 36: Vergleich des Parameters „Start Render" bei Anwendung von vielen aufgeteilten Dateien und wenigen großen Dateien unter dem HTTP/2 – Protokoll bei 3G-Verbindung.

„Visual Progress“

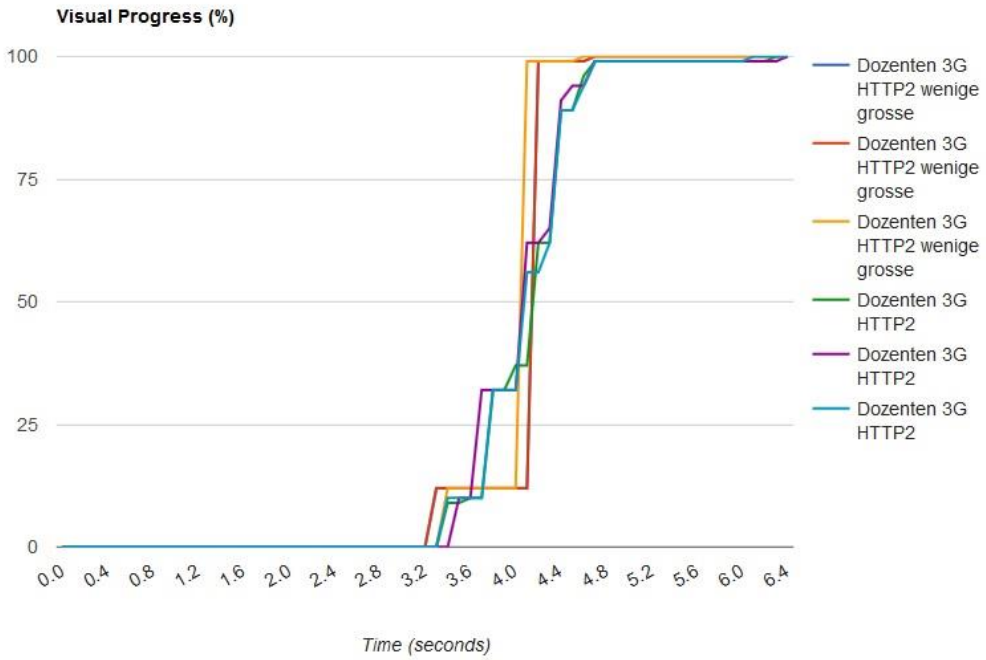

Abb. 54: „Visual Progress“ bei Anwendung von vielen aufgeteilten Dateien und wenigen großen Dateien unter dem HTTP/2 – Protokoll bei 3G-Verbindung.

Aus der oben dargestellten Grafik sieht man, dass in beiden Anwendungen die ersten Pixel fast gleichzeitig auf dem Viewport angezeigt wurden. Einen größeren Unterschied sieht man, beim Ladefortschritt der Inhalte der Webapplikation. Bei der Anwendung mit vielen kleinen Dateien werden die Inhalte stetig nacheinander geladen. Im Gegenteil werden die Inhalte bei der Anwendung mit wenigen großen Dateien in einem großen Sprung geladen. Dies liegt an dem einen großen Bild.

Aus den aufgenommenen Screenshots kann man sehen, wie die Ressourcen geladen werden. In dem Moment, in dem das große Bild der Anwendung mit wenigen großen Dateien angezeigt wird, ist schon etwa die Hälfte der kleineren Bilder der zweiten Anwendung erschienen. Für das User Experience ist in diesem Fall die Anwendung mit vielen aufgeteilten Ressourcen besser, da die Nutzer schon etwa 400 ms früher erste Bilder auf der Seite sehen können.

Das verwunderliche ist, dass die Anwendung mit vielen kleineren Dateien deutlich mehr Zeit braucht, um alle Ressourcen herunterzuladen. Aus den Screenshots sieht man, dass der Unterschied bei 1,6 Sekunden liegt.

5.8.1. Zwischenfazit: Bestätigt sich die Überlegung, nach der aufgeteilte kleinere Dateien für das HTTP/2 – Protokoll besser sind?

Im Laufe des Tests wurde festgestellt, dass die Anwendung mit vielen kleinen Dateien unter mobilen Netzwerken mehr Zeit zum Laden braucht als die Anwendung mit wenigen großen

Dateien. Dies sieht man auch bei den Unterschieden zwischen dem Parameter „Start Render". Was könnte der Grund dafür sein?

Die Antwort auf die Frage, warum kleinere Dateien unter dem HTTP/2 – Protokoll bei hoher Latenz länger zum Herunterladen brauchen, als größere Dateien, lässt sich nicht einfach finden. Deshalb wird die Dozenten-Seite unter gleichen Bedingungen noch am lokalen PC mithilfe vom Browser „Mozilla Firefox", Version 47.0 aufgerufen und einzelne TCP – Pakete in „Wireshark" genauer untersucht. Im Browser „Mozilla Firefox" gibt es keine „Throttling"-Funktion oder kein Add-on, mithilfe dessen man die Verbindung simulieren kann und die Latenz steuern kann. Deshalb wird unter Kabel-Verbindung geschaut, wie die Pakete dem Client zugestellt werden.

Mit der Aufnahme von „Wireshark" kann man genauer sehen, wie die Kommunikation zwischen dem Client und dem Server stattfindet. Zuerst werden die Ressourcen mithilfe von HEADERS – Frames vom Client angefragt. Abb.55 veranschaulicht den Kommunikationsprozess zwischen dem Client und dem Server unter dem HTTP/2 – Protokoll. Es werden mehrere Ressourcen nacheinander angefragt. Danach schickt der Server die Antworten an den Client. Dies passiert in folgender Reihenfolge: zuerst kommt der HEADERS – Frame, gefolgt vom DATA – Frame der als erstes vom Client angefragten Datei. Danach erreicht die nächste Ressource den Client. Deshalb kann man sagen, dass die Ressourcen den Client in der FIFO (First In-First Out) – Reihenfolge erreichen. Nachdem die zuerst angefragten Dateien den Client erreichen, wird die nächste „Ressourcenportion" vom Server angefragt. Dies bedeutet, dass mehrere angefragte Dateien gleichzeitig am Server bearbeitet werden. Manchmal werden mehrere HEADERS- oder DATA – Frames nacheinander geliefert.

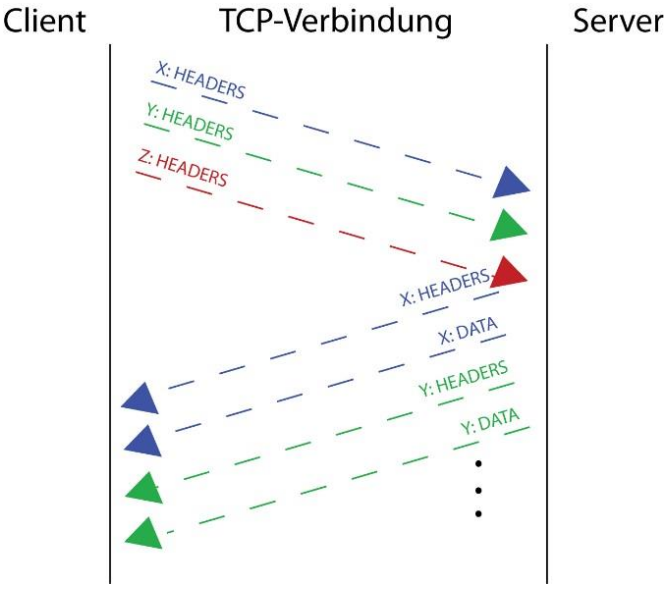

Abb. 55: Schematische Darstellung des Kommunikationsprozesses zwischen dem Client (Mozilla Firefox 47.0) und dem Server (Apache 2.4.20) unter dem HTTP/2 – Protokoll

Woran könnte es liegen, dass unter dem HTTP/1.1 – Protokoll die Dateien schneller geladen werden, als unter dem HTTP/2 – Protokoll?

Im aktuellen Test wurde festgestellt, dass für den Parameter „Start Render" unter dem HTTP/2 – Protokoll die Webapplikation mit aufgeteilten Dateien etwas besser ist, weil kleinere Bilder schneller im Browser erscheinen. Allerdings sind die Parameter „Start Render" und „domComplete" bei Anwendung von aufgeteilten Dateien langsamer. Auch in einem Test zuvor wurde festgestellt, dass die Webapplikation mit aufgeteilten Ressourcen unter dem HTTP/2 – Protokoll später im Browser angezeigt wird, als unter dem HTTP/1.1 – Protokoll mit zusammengefassten Dateien.

Es ist interessant, die Ergebnisse des aktuellen Tests mit den Ergebnissen des Tests zuvor („Untersuchung der Dozenten-Seite unter dem HTTP/1.1- und HTTP/2 – Protokoll, unter 3G-Verbindung") zu vergleichen. Wenn man die aufgenommenen Screenshots dieser Tests miteinander vergleicht, ist zu sehen, dass alle Dateien, egal ob sie aufgeteilt oder zusammengefasst sind, unter dem HTTP/2 – Protokoll länger zum Laden brauchen als unter dem HTTP/1.1 – Protokoll. Dazu kommt noch, dass der Parameter „Start Render" unter dem HTTP/2 – Protokoll auch in beiden Fällen später ausgegeben wird. Dies bedeutet, dass nicht nur eine größere Anzahl an Ressourcen, die durch genau eine TCP – Verbindung läuft, eine entscheidende Rolle an der Parameterverzögerung spielt.

Es gibt noch eine Vermutung, warum die Webapplikation unter dem HTTP/2 – Protokoll langsamer ist, als beim HTTP/1.1 – Protokoll. Es sieht so aus, als ob die Multiplexierung von „Frames" nicht perfekt funktioniert. Zuerst kommt eine „Portion" von HEADERS - Frames zum Server, der nachher in der Reihenfolge „First In-First Out" die Ressourcen zurück liefert (siehe Abb. 107). Außerdem ist selten zu sehen, dass zwischen HEADERS – Frame und DATA – Frame eines „Streams" noch die „Frames" von einem anderen „Stream" reinkommen. Dadurch sieht man, dass die „Frames" nicht wirklich durchmischt sind. Es wäre wahrscheinlich besser, wenn die Requests und Responses von unterschiedlichen „Streams" besser durchmischt werden, damit der Server nicht lange auf Anfragen warten muss und diese dann gleichzeitig bearbeiten muss (siehe Abb. 55).

Ein weiterer Grund könnte sein, dass trotz der Verbindungsaufbauzeit zum Server, sechs oder mehr Verbindungen die Ressourcen schneller an den Client liefern können, als nur eine TCP – Verbindung unter dem HTTP/2 – Protokoll.

Es bleibt aber schwierig, eine genaue Aussage darüber zu machen. Auf der Dokumentationsseite des Apache-Moduls, das das HTTP/2 – Protokoll implementiert, steht, dass sich dieses Modul im experimentellen Stadium befindet (Apache Software Foundation 2016a). Daher ist schwer zu sagen, wie genau der Browser die Responses bearbeitet. Die Frage, warum die Webapplikation unter dem verschlüsselten HTTP/2 – Protokoll mehr Zeit braucht, um erste

Pixel anzuzeigen und die Ressourcen herunterzuladen, als unter dem unverschlüsselten HTTP/1.1 – Protokoll, bleibt noch offen.

5.9. Wie funktioniert das HTTP/2 – Protokoll in anderen Browsern?

Bisher wurden alle Tests mit dem Client „Mozilla Firefox" (Version 47.0) gemacht. Es ist interessant, zu beobachten, wie andere Browser die HTTP/2 – Implementierung bearbeiten. Laut den „StatCounter Global Stats" Statistiken (siehe Anhang 9.2) , gibt es im Wesentlichen drei Browser weltweit, die zwischen August 2015 und Juli 2016 auf Desktop-PCs benutzt werden: „Chrome" mit fast 60% Nutzeranteil, „Firefox" mit fast 16% und „Internet Explorer" (Version 11) mit 10,5% (Anhang 9.2) (StatCounter 2016). Deshalb wird das HTTP/2 – Protokoll unter den Browsern „Chrome" (Version 52.0) und „Internet Explorer" (Version 11) getestet. Außerdem wird geschaut, wie diese Browser per „Server Push" übergebene Ressourcen bearbeiten.

Um zu vergleichen, wie schnell diese Browser die Webapplikation unter dem HTTP/2 – Protokoll auf dem Viewport anzeigen, wird der Parameter „Visual Progress" für jeden Browser betrachtet. Danach werden die Ergebnisse der Implementierung des HTTP/2 – Protokolls bei diesen Browsern verglichen.

5.9.1. Test 1: Gibt es deutliche Unterschiede in der Bearbeitungsart des neuen Protokolls zwischen den drei populärsten Desktop-Browsern?

Testbedingungen: als Testseite dient die Dozenten-Seite der Webapplikation. Die Webapplikation wurde jeweils neun Mal mit den drei Browsern mithilfe von „Webpagetest.org" aufgerufen.

Zum Vergleich wird der Medianwert verwendet (von „Webpagetest.org" ausgewählter Wert, der sich an dem Parameter „Start Render" orientiert).

Aus Abb. 56 ist zu sehen, dass der Parameter „Visual Progress" unter „Mozilla Firefox" und „Microsoft IE" ähnlich ist. Nun kann man sagen, dass in „Microsoft IE" die Webapplikation etwas früher angezeigt wird und die meisten Inhalte (bei fast 100%) den Client etwas früher erreichen werden. Gleichzeitig sieht man, dass die Inhalte unter „Google Chrome" deutlich später angezeigt werden. Interessant zu bemerken ist, dass bis 4,4 Sekunden nur etwa 20% des Inhalts angezeigt wird. Um diese Ergebnisse besser verstehen zu können, wird geschaut, wie diese Browser das HTTP/2 – Protokoll verarbeiten.

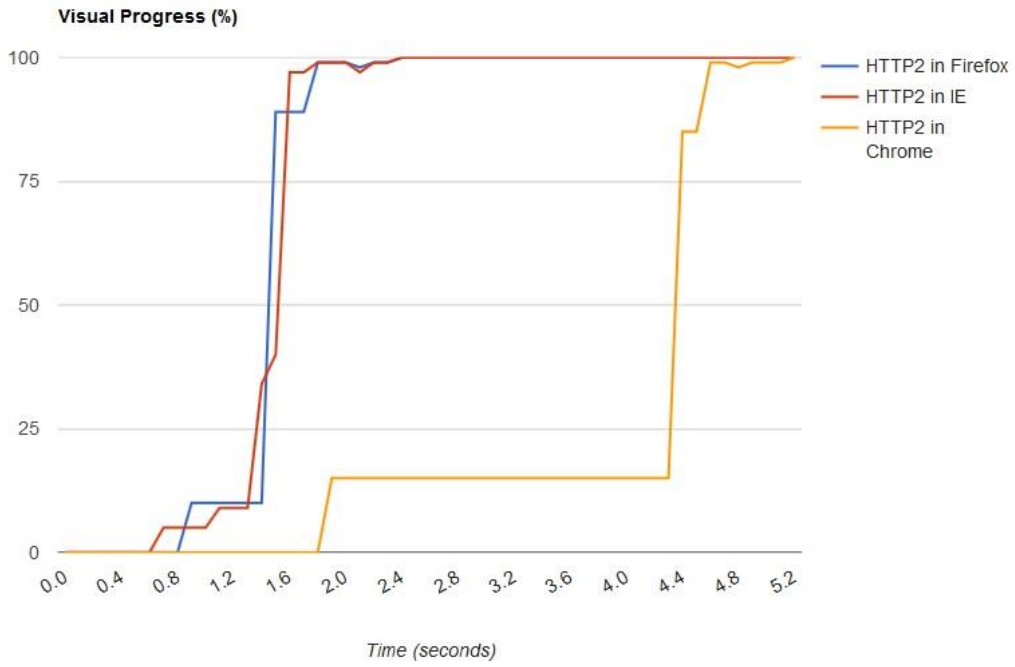

Abb. 56: „Visual Progress" in Mozilla Firefox v.47.0, Microsoft IE11 und Google Chrome v.52.0 unter dem HTTP/2 – Protokoll und Kabel-Verbindung.

Wenn man den Ressourcenwasserfall unter „Microsoft IE11" anschaut, sieht man, dass das HTTP/2 – Protokoll als HTTP/1.1 – Protokoll mit TLS – Verschlüsselung interpretiert wird. Dies bedeutet, dass der Client („Microsoft IE11") das HTTP/2 – Protokoll nicht versteht und die Kommunikation beim HTTP/1.1 - Protokoll geblieben ist. Zur Prüfung wurde die Webapplikation mit dem gleichen Browser am lokalen PC aufgerufen. Die Ergebnisse waren die gleichen. „Microsoft IE" blieb beim HTTP/1.1 – Protokoll mit TLS – Verschlüsselung.

Die Kommunikation zwischen dem Server und dem Browser „Google Chrome" läuft jedoch als HTTP/2 – Protokoll. Der Browser „Google Chrome" akzeptiert das HTTP/2 – Protokoll. Es ist allerdings sehr verwunderlich, dass der Unterschied der Ladezeit der Webapplikation zwischen „Mozilla Firefox" und „Google Chrome" sehr groß ist. Wenn man den Ressourcenwasserfall bei „Google Chrome" betrachtet, ist auffällig, dass die Ressourcen, wie beim HTTP/1.1 – Protokoll, nacheinander geladen werden. Aus dem „Connection View" (im „Webpagetest.org" - Tool) ist zu sehen, dass es innerhalb einer TCP – Verbindung viele leere Stellen gibt, an denen der Server keine Inhalte liefert (Abb. 57).

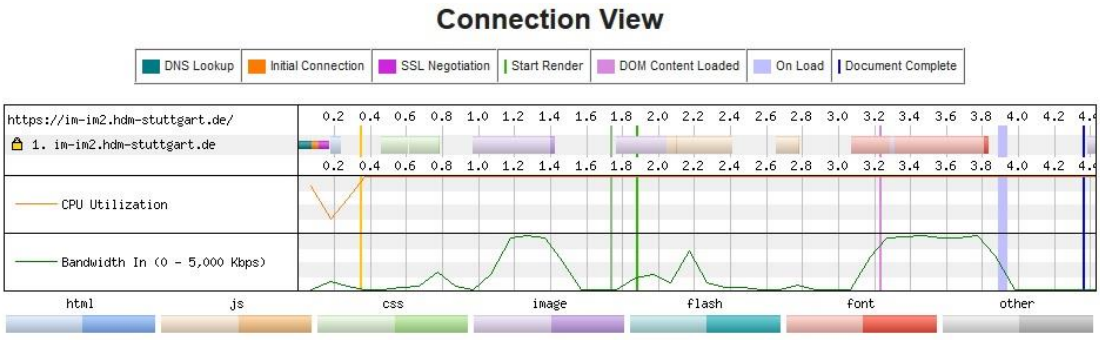

Abb. 57: „Connection View" für die Startseite beim „Google Chrome"-Client

Es ist nicht klar, was die Client-Seite an den leeren Stellen macht. Wenn man den Ressourcenwasserfall mit dem „Connection View" vergleicht, ist zu beobachten, dass, nachdem eine „Portion" von Ressourcen geladen wird, mehrere hundert Millisekunden nichts passiert. Danach wird die nächste „Portion" Ressourcen vom Client angefragt und geliefert.

Der Frame-Austausch zwischen dem Client „Google Chrome" mit Version 52.0 und dem Server unter dem HTTP/2 – Protokoll soll genauer betrachtet werden. Dafür wird die Webapplikation am lokalen PC aufgerufen und mithilfe von „Wireshark" genauer untersucht.

Anhand der Mitschnitte sieht man, dass die Ressourcen nacheinander aufgerufen und an den Client geliefert werden. Dieser Datenmitschnitt ist sehr ähnlich zum Datenmitschnitt aus dem Browser „Mozilla Firefox". Es ist allerdings zu bemerken, dass die Frames-Multiplexierung im Browser „Google Chrome" scheinbar etwas besser funktioniert.

Nach dem Aufruf der Webapplikation am lokalen PC durch den Browser „Google Chrome" wurde die „.har" Datei gespeichert. Wenn man diese Datei in einem „Har Viewer" ansieht (Hemming 2016), (Duran 2016), sieht man den gesamten Ressourcenwasserfall. Hier sieht man eine ganz andere Ressourcenauslieferung als im „Webpagetest.org" – Tool. Alle Ressourcen werden einigermaßen gleichzeitig angefragt, ohne dass leere Stellen beobachtet werden können. Dies zeigt, dass sich die Funktionsweise des Browsers „Google Chrome" am lokalen PC und im „Webpagetest.org" – Tool stark unterscheiden.
In diesem Fall ist es interessant, die Ressourcenwasserfälle aus „Google Chrome" und „Mozilla Firefox", die am lokalen PC aufgenommen wurden, miteinander zu vergleichen. Dies kann man mithilfe eines „Har-Viewer" machen (Hemming 2016), (Duran 2016). Es ist zu sehen, dass Ressourcen, welche durch „Mozilla Firefox" aufgerufen wurden, etwas schneller geladen werden.

5.9.2. Zwischenfazit: Gibt es deutliche Unterschiede in der Bearbeitungsart des neuen Protokolls zwischen den drei populärsten Desktop-Browsern?

Aus den oben dargestellten Ergebnissen kann man sagen, dass jeder Browser auf seine eigene Weise das HTTP/2 – Protokoll bearbeitet. Es wurde festgestellt, dass der Client „Microsoft IE11" das HTTP/2 – Protokoll nicht versteht. Neuere Versionen dieses Browsers können die existierende HTTP/2 – Implementierung evtl. anders interpretieren.

Die zweite Feststellung war die, dass der Browser „Google Chrome“ das HTTP/2 – Protokoll ganz anders bearbeitet, als der Browser „Mozilla Firefox“. Allerdings unterscheiden sich die Ergebnisse des „Webpagetest.org“ – Tool von denen am lokalen PC. Aber in beiden Fällen sieht man, dass die Webapplikation, die durch „Mozilla Firefox“ aufgerufen wurde, schneller geladen wurde. Es ist schwer zu sagen, auf welche Art die Browser die Ressourcen bearbeiten. Es wird aber wohl noch etwas Zeit brauchen, bis alle Browser das HTTP/2 – Protokoll richtig interpretieren werden.

5.9.3. Test2: Wie wird der Browser „Google Chrome“ per „Server Push“ übergebene Ressourcen interpretieren?

Im Unterkapitel „Wie funktioniert Server Push?“ wurde die Funktionsweise des „Server Pushs“ erklärt und alle dazugehörigen Tests wurden mithilfe des Browsers „Mozilla Firefox“ gemacht. Im zuvor gemachten Test wurde festgestellt, dass der Browser „Microsoft IE11“ das HTTP/2 – Protokoll nicht bearbeiten kann. Aus diesen Gründen wurde im aktuellen Test nur geschaut, wie „Google Chrome“ gepushte Ressourcen bearbeitet.

*Testbedingungen***:** die Dozenten-Seite der Webapplikation wird untersucht. Es wurden 10 CSS Dateien per „Server Push“ übergeben. Die Webapplikation wurde neun Mal unter „Mozilla Firefox“ und „Google Chrome“ mithilfe von „Webpagetest.org“ aufgerufen. Zusätzliche Untersuchungen unter „Mozilla Firefox“ dienen dazu, um die Ergebnisse mit „Google Chrome“ vergleichen zu können.

Zum Vergleich dient ein von „Webpagestest.org“ ermittelter Medianwert.

Aus Abb. 58 sieht man, dass die Unterschiede zwischen der Zeit der Erstdarstellung und der Darstellung der meisten Inhalte auf dem Viewport zwischen beiden Browsern sehr groß ist. Der Unterschied zwischen der Zeit der Erstdarstellung beträgt etwa 3,5 Sekunden. Aus dem Ressourcenwasserfall sieht man aber, dass „Server Push“ innerhalb des Browsers „Google Chrome“ akzeptiert wird und die Ressourcen wie erwartet den Client erreichen.

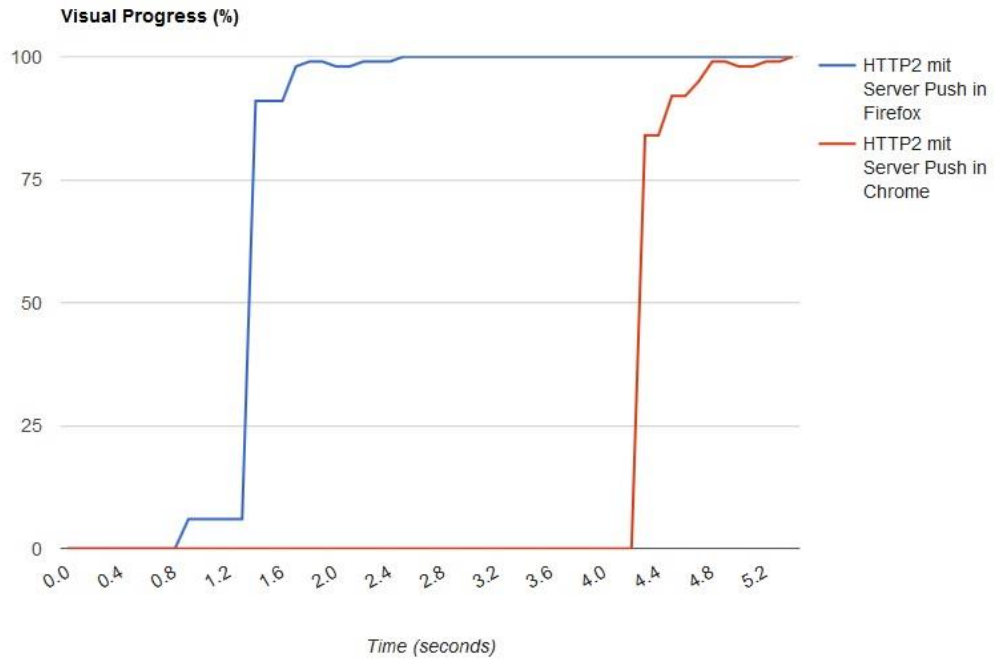

Abb. 58: „Visual Progress" in „Mozilla Firefox" v.47.0, und „Google Chrome" v.52.0 unter dem HTTP/2 – Protokoll mit per „Server Push" übergebenen CSS Dateien bei Kabel-Verbindung.

Der vorangegangene Test hat gezeigt, dass die Ressourcen unter dem HTTP/2 – Protokoll mithilfe des Browsers „Google Chrome" deutlich länger geladen werden als unter dem Browser „Mozilla Firefox". Es wurde bemerkt, dass große Unterschiede nur anhand der Daten des „Webpagetest.org" – Tools erkannt werden können. Aus diesem Grund wird die Webapplikation am lokalen PC mithilfe von „Wireshark" zusätzlich untersucht.

Es wird der Browser „Google Chrome" v. 52.0 verwendet. Während die Ressourcennachladung mithilfe der Entwicklertools im Browser untersucht worden ist, wurde bemerkt, dass gepushte CSS Dateien noch einmal nachgefragt werden. Dies sieht man aus dem Ressourcenwasserfall. Dies passiert, nachdem alle Bilder und JavaScript Dateien fertig geladen wurden. Dies kann nur bedeuten, dass alle per „Server Push" übergebenen Ressourcen von der verwendeten Browserversion nicht vollständig akzeptiert wurden und deshalb noch einmal nachgefragt werden.

Die gleiche Vorgehensweise sieht man im Datenmittschnitt von „Wireshark". Wenn man in „Wireshark" den Filter auf eine der gepushten CSS Dateien anwendet: *„http2.header.value == "/css/style.css""*, sieht man, dass diese zuerst mit einem PUSH_PROMISE – Frame angekündigt wird (siehe das Beispiel auf dem beigelegten Datenträger unter „Unterschiedliche Browser\HTTP2 Server Push_CSS\Google Chrome\Aus dem Arbeits-PC\Server Push Chrome .pcapng"). Wenn man den dazugehörigen „Stream" mit dem Filter *„http2.Streamid == 2"* verfolgt, sieht man, dass diese Datei dem Client zugestellt wird. Danach wird diese Datei jedoch nochmal am Server angefragt und dem Client zugestellt. Man kann nur einen Unterschied zwi-

schen der Größe des TCP – Paketes erkennen, in dem HEADERS- und DATA – Frames zusammen übertragen wurden. Wenn die Daten dieser Datei durch „Server Push" zugestellt wurden, beträgt die Größe des TCP – Paketes 292 Bytes. Wenn die Datei vom Browser angefragt wird, hat das TCP – Paket vom Server eine andere Größe von HEADERS- und DATA – Frames, nämlich 1506 Bytes.

Daher kann man sagen, dass die vom Server zuerst gelieferten Daten nicht vollständig am Client ankommen. Deshalb frägt der Client diese Dateien noch Mal an.

5.9.4. Zwischenfazit: Wie wurden per „Server Push" übergebene Ressourcen auf der Client-Seite durch den Browser „Google Chrome" bearbeitet?

In einem zuvor durchgeführten Test wurde festgestellt, dass das HTTP/2 – Protokoll unter „Google Chrome" anders bearbeitet wird, als bei „Mozilla Firefox". Im aktuellen Test wurde besonders darauf geachtet, wie der Browser per „Server Push" übergebene Ressourcen bearbeitet. Man sieht wieder, dass, obwohl die Browserversionen beim „Webpagetest.org" – Tool und am lokalen PC die gleichen sind, sie ganz unterschiedliche Ergebnisse zeigen. Aus den Ergebnissen des „Webpagetest.org" – Tools ist zu bemerken, dass der „Server Push" - Einsatz wie gewöhnlich funktioniert. Anders sehen die Ergebnisse aus, die am lokalen PC aufgenommen wurden. Bei der Untersuchung am lokalen PC wurde festgestellt, dass per „Server Push" übergebene Ressourcen den Client nicht vollständig erreichen. Deshalb wurden gepushte Ressourcen noch Mal am Server angefragt und dem Client geliefert.

Diese Ergebnisse sind schwer zu erklären. Vermutlich braucht es noch etwas Zeit, bis die Implementierung des neuen Protokolls von den verschiedenen Browsern richtig interpretiert wird.

5.10. Angetroffene Schwierigkeiten während der Testdurchführung und der Testevaluation

Während der Testdurchführung und Testauswertung wurden mehrere Schwierigkeiten getroffen. Diese werden hier kurz zusammengefasst.

Funktionsweisen der Browser des „Webpagetest.org" – Tool und des lokalen PCs unterscheiden sich sehr.

Für die Durchführung der Tests wurde vor allem das online-Tool von „Webpagetest.org" benutzt. Es wurde die Funktionsweise des HTTP/2 – Protokolls geprüft. Wenn die Webapplikation zur Prüfung am lokalen PC aufgerufen wurde, wurde festgestellt, dass die Reihenfolge der Ressourcen sich von der Reihenfolge der Ressourcen aus dem „Webpagetest.org" – Tool unterscheidet, obwohl die Webapplikation unter gleichen Browserversionen („Mozilla Firefox" mit Version 47.0) aufgerufen wurde. Solches Verhalten ist auch beim Browser „Google Chrome" mit Version 52.0 zu treffen. Wenn der Slider und die dazugehörigen Bilder später aufgerufen werden, werden die Inhalte später auf dem Viewport angezeigt. Deshalb wird die Webapplikation später den Parameter „Visual Progress" 100% erreichen.

Es wurde geprüft, ob per „Server Push" übergebene Ressourcen beim zweiten Aufruf der Webapplikation im Browser-Cache landen. Als diese Vorgehensweise am lokalen PC mithilfe des Browsers „Mozilla Firefox" geprüft wurde, wurde festgestellt, dass alle gepushten Ressourcen vom Server nochmal aufgerufen werden, während die anderen Ressourcen im Browser-Cache gelandet sind. Andersrum werden im „Webpagetest.org" – Tool beim zweiten Aufruf der Webapplikation keine Dateien mehr geladen.

Fehlende Implementierungen, Beispiele und Evaluationen des HTTP/2 – Protokolls unter dem Apache Server.

Zu der Zeit, als die Tests durchgeführt wurden, gab es keine offenen Implementierungen und Beispiele zur Anwendung des HTTP/2 – Protokolls unter dem Apache Server. Es wurden nur wenige ganz konkrete Implementierungen gefunden, die vor allem bei der Anwendung anderer Server („Nghttp2", „NGINX" oder „H2O") gemacht wurden.

Einer der wenigen Leitfäden für die Serverkonfiguration des verwendeten Servers war die offizielle Dokumentation des Apache-Moduls „mod_http2" ab Version 2.4.17. (Apache Software Foundation 2016a). Leider hat die Anwendung der Einstellungen, die für die Priorisierung der per „Server Push" übergebenen Ressourcen zuständig sind, nichts verändert (siehe den Teil: „Serverseitige Priorisierung von per „Server Push" übergebenen Ressourcen"). Die einzige Möglichkeit, um die Bandbreite zwischen gepushten Ressourcen zu kontrollieren, ist die Reihenfolge, in der die Ressourcen im <VirtualHost> per „Header" aufgelistet sind.

Empfehlungen zu den clientseitigen Performance – Optimierungstechniken für das HTTP/2 – Protokoll stimmen nicht immer mit den praktischen Ergebnissen überein.

Die Funktionsweise des HTTP/2 – Protokolls und mögliche Optimierungstechniken unter diesem Protokoll wurden oftmals allgemein gefasst und nur in der Theorie beschrieben. In der Praxis funktioniert jeder Server auf eigene Art und deshalb kann nicht immer genau gesagt werden, welche Optimierungstechniken sich auf jeden Fall gut eignen. Die gesamte Funktionsweise hängt auch von einem konkreten Browser ab und kann sich je nach Version unterscheiden. Z.B. wurde in einem oben durchgeführten Test festgestellt, dass, im Gegensatz zu den theoretischen Erkenntnissen, die Webapplikation langsamer wird, je mehr kleine Dateien unter dem HTTP/2 – Protokoll vom Server heruntergeladen werden.

Nicht alle der populärsten Browser können das neue Protokoll korrekt interpretieren.

Wie zuvor durch Tests festgestellt wurde, wird das HTTP/2 – Protokoll in den populärsten Browsern sehr unterschiedlich verarbeitet. Der Browser „Microsoft IE11" interpretiert das neue Protokoll als HTTP/1.1 – Protokoll mit TLS – Verschlüsselung. In den Tests, in denen das HTTP/2 – Protokoll unter dem Browser „Google Chrome" getestet wurde, wurde festgestellt, dass es sehr unterschiedlich interpretiert wurde. Die Testergebnisse aus dem „Webpagetest.org" – Tool zeigen, dass das HTTP/2 – Protokoll nicht richtig verarbeitet wird: es gibt viele Stellen, an denen

nichts zwischen dem Browser und dem Server gemacht wird. Wenn die Funktionsweise des HTTP/2 – Protokolls am lokalen PC mit der gleichen Browserversion getestet wurde, wurde festgestellt, dass die Ressourcen im Vergleich zum Browser „Mozilla Firefox" mehr Zeit brauchen, um geliefert zu werden.

Beim „Server Push"-Test im Browser „Google Chrome" (Version 52.0), wurde festgestellt, dass diese Funktion von diesem Browser nicht richtig bearbeitet wird. Die Testergebnisse am lokalen PC zeigen, dass alle gepushten Dateien zusätzlich vom Client angefragt werden, obwohl diese schon nach der HTML Datei an den Client geliefert wurden. Dem Datenmitschnitt aus „Wireshark" nach zu urteilen, werden per „Server Push" übergebene Dateien nicht vollständig geliefert. Im Browser „Mozilla Firefox" ist diese Vorgehensweise nicht zu beobachten.

„Wireshark" versteckt manche „Frames", die sich innerhalb eines TCP – Paketes befinden.

Mithilfe von „Wireshark" kann man alle Frames des HTTP/2 – Protokolls genauer untersuchen. Der Filter „http2" zeigt die gesamte Kommunikation zwischen Client und Server (siehe Abb. 59). In der letzten Spalte des oberen Bereichs („Info") sieht man, welche Frames in einem TCP – Paket zusammengefasst wurden. Allerdings kann man manchmal Situationen treffen, in denen nicht alle übertragenen „Frames" in dieser Spalte angezeigt werden. Abb. 59 veranschaulicht, dass zuerst HEADERS- und DATA – Frames des 20. "Streams" vom Server geliefert werden. Im gleichen TCP – Paket wird aber nachher noch ein HEADERS – Frame des 22.-"Streams" geliefert. Dieser ist aber nicht in der Spalte „Info" mit den anderen „Frames" zu sehen. Deshalb ist es manchmal etwas schwierig, um einzelne „Frames" zu verfolgen. Aber mithilfe von Filtern kann man immer nach einzelnen „Streams" und dazugehörigen „Frames" suchen.

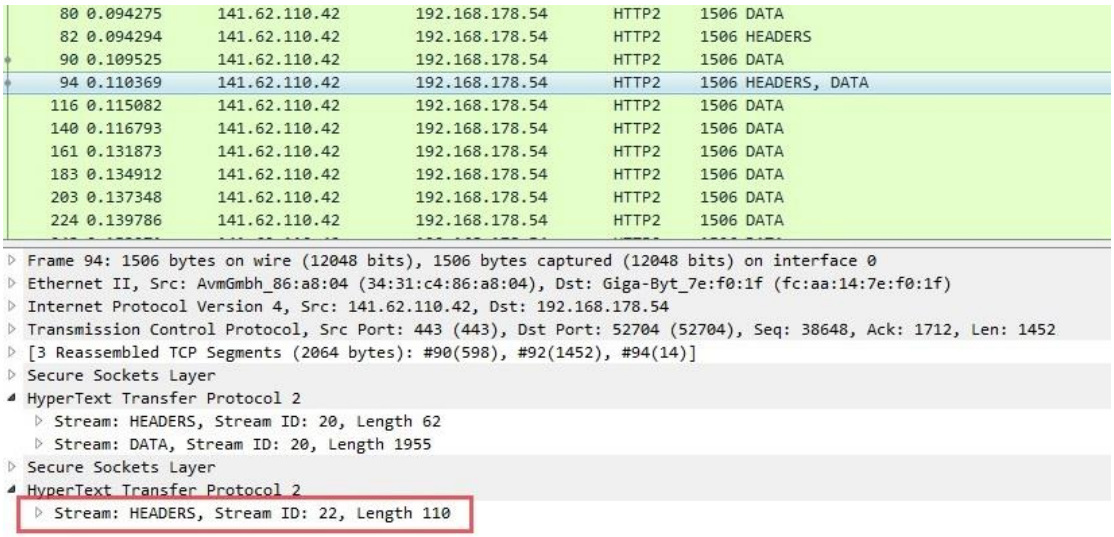

Abb. 59: Screenshot aus „Wireshark": Datenmitschnitt des HTTP/2 –Protokolls.

6. Fazit und Ausblick

Abschließend sollen noch einmal die wichtigsten Schritte, Aussagen und Schlussfolgerungen zusammengefasst werden.

Innerhalb dieser Masterarbeit wurden hauptsächlich drei Ziele verfolgt:

- Untersuchung der Ladezeit der Webapplikation mithilfe der Funktion des HTTP/2 – Protokolls „Server Push" unter unterschiedlichen Einsätzen. Besonders geachtet wurde dabei auf clientseitige Frontend Performance – Optimierungstechniken.
- Allgemeine Untersuchung der Ladezeit der Webapplikation unter dem HTTP/1.1- und dem HTTP/2 – Protokoll, wenn die Webapplikation gesondert für jede Version des HTTP – Protokolls optimiert wurde.
- Auseinandersetzung mit der Konfiguration des Webservers für das HTTP/2 – Protokoll im Hinblick auf den „Server Push" und mit den wichtigsten Schritten und Parametern des Ladezeitprozesses.

Alle Schritte zum Erreichen der gestellten Ziele wurden abgeschlossen. Im theoretischen Teil dieser Masterarbeit hat sich herausgestellt, dass im Vergleich zum HTTP/1.1 – Protokoll das HTTP/2 – Protokoll die Inhalte der Webapplikationen schneller, effizienter und ohne zusätzliche Latenz übertragen soll. Es wurden bestimmte clientseitige Techniken zur Performance – Optimierung für beide Protokolle definiert.

Nach diesem Schritt wurden bestimmte Parameter des kritischen Rendering – Pfades ausgewählt und mit Tools zur Messung der Performance („Webpagetest.org") und zur Netzwerkanalyse („Wireshark") untersucht.

Es gab drei wesentliche Einsätze für die Testausführung. Zuerst wurde untersucht, wie unter dem HTTP/2 – Protokoll die Funktion „Server Push" funktioniert. Gepushte Ressourcen erreichen den Client genau in der Reihenfolge, wie diese auf den serverseitigen Einstellungen aufgelistet sind. Allerdings wurde festgestellt, dass per „Server Push" übergebene Ressourcen nicht priorisiert werden, auch wenn Priorisierungen für einzelne Ressourcentypen serverseitig eingestellt wurden. Deshalb kann festgestellt werden, dass innerhalb der ausgewählten server- und clientseitigen Bedingungen die Eigenschaft der Priorisierungen von gepushten Ressourcen funktionsunfähig ist.

Außerdem wurde geprüft, ob gepushte Ressourcen nach dem ersten Aufruf im Browser Cache landen werden. Es wurde festgestellt, wenn die Ressourcen vom lokalen PC aufgerufen werden, werden diese bei den weiteren Aufrufen immer wieder geladen (siehe Unterkapitel „Wie funktioniert 'Server Push'? "). Dies bedeutet, dass das Browsercaching für gepushte Ressourcen nicht funktioniert. Dies wiederspricht den theoretischen Erkenntnissen (siehe Unterkapitel „3.3 Vorstellung des HTTP/2 – Protokolls"). Allerdings unterscheiden sich die Ergebnisse aus

dem „Webpagetest.org“ – Tool, in dem beim zweiten Aufruf per „Server Push“ übergebene Ressourcen im Browser Cache landen.

Danach wurde der „Server Push“ in Bezug auf den kritischen Rendering – Pfad untersucht. Es wurde besonders auf früher gewählte Metriken während der Ladezeit geachtet und es wurden Einsatzmöglichkeiten gesucht, die für den kritischen Rendering – Pfad vorteilhaft sind. Es wurde festgestellt, dass sich der „Server Push“ – Einsatz für alle für die Erstdarstellung kritische Ressourcen gut eignet. Auch wenn nicht – kritische, aber aus der Nutzersicht wichtige Ressourcen (Webschriften oder Bilder) gepusht werden, können auch gute Ergebnisse entstehen. Allerdings muss aufgepasst werden, dass die nicht – kritischen Ressourcen keine wichtigen CSS Dateien blockieren werden und dass nicht zu viele Ressourcen gepusht werden (siehe Unterkapitel „5.5 Untersuchungen zum ‘Server Push’ “). Sonst kehrt das „Header-Of-Line Blocking“ – Problem zurück. Die Ausnahmen sind unkritische JavaScript – Dateien, die zusätzlich wegen deren sofortigen Ausführung alle anderen Ressourcen blockieren werden.

Danach wurde die allgemeine Ladezeit der Webapplikation unter dem unverschlüsselten HTTP/1.1- und dem HTTP/2 – Protokoll verglichen. Obwohl die Zeit zur Netzwerkverbindung unter dem HTTP/2 – Protokoll kleiner oder fast gleich als unter dem HTTP/1.1 – Protokoll war, wurde die Ladezeit unter der neueren Version des HTTP – Protokolls deutlich langsamer. Unter mobilen Netzwerken waren die Unterschiede besonders groß. In der Einleitung dieser Masterarbeit wurde erwähnt, dass etwa 80-90% der gesamten Geschwindigkeit der Webapplikation am Frontend liegt (Rigor, Inc. 2016). Wie die Testergebnisse gezeigt haben, genügen für die Erstellung einer perfomanten Webapplikationen die clientseitigen Performance – Optimierungstechniken für das HTTP/2 – Protokoll nicht. Um performante Webapplikationen unter dem neuen Protokoll zu erreichen, muss noch das TCP – Protokoll beachtet werden, weil nur eine TCP – Verbindung zwischen Client und Server existiert. Dazu müssen der Traffic, die Reihenfolge von TCP – Paketen und deren Inhalte genau beobachtet werden.

In einem weiteren Test wurde untersucht, ob die Aufteilung der Ressourcen unter dem HTTP/2 – Protokoll eine Rolle spielt. Im theoretischen Kapitel wurde festgestellt, dass dank der Multiplexierung von Requests und Responses innerhalb der bestehenden TCP – Verbindung die Ressourcen nicht zusammengefasst werden müssen, wie es für das HTTP/1.1 – Protokoll gemacht wird (Grigorik 2013, 243), (NGINX, Inc. 2015, 5). Allerdings wurde dabei festgestellt, sollten Ressourcen nicht zusammengefasst werden, wird die Ladezeit der Webapplikation deutlich verlangsamt. Dies wiederspricht den theoretischen Erkenntnissen (siehe Unterkapitel „3.4 Mögliche Optimierungstechniken für das HTTP/2 – Protokoll“).

Im nächsten Test wurde untersucht, wie das neue Protokoll in unterschiedlichen Browsern funktioniert. Es wurden die drei meist verwendeten Browser genommen: „Mozilla Firefox“, „Google Chrome“ und „Microsoft IE 11“. Tests wurden sowohl am lokalen PC als auch mit dem „Web-

pagetest.org“ – Tool gemacht. Es hat sich herausgestellt, dass sich die Testergebnisse stark unterscheiden. Im Browser „Google Chrome“ innerhalb des „Webpagetest.org“ – Tools wurde die Webapplikation im Vergleich zum Browser „Mozilla Firefox“ sehr langsam geladen. Die Gründe dafür bleiben ungeklärt. Wenn die Ladezeit zwischen diesen Webbrowsern am lokalen PC verglichen wurde, wurde festgestellt, dass unter dem Browser „Mozilla Firefox“ die Webapplikation schneller geladen wird. Im Browser „Microsoft IE 11“ wird die Kommunikation zwischen dem Client und dem Server immer unter dem HTTP/1 – Protokoll laufen.

Im letzten Test wurde geschaut, wie der Browser „Google Chrome“ per „Server Push“ übergebene Ressourcen bearbeitet. Tests wurden sowohl am lokalen PC als auch mit dem „Webpagetest.org“ – Tool gemacht. Es hat sich herausgestellt, dass sowohl im Browser „Google Chrome“ als auch im Browser „Mozilla Firefox“ innerhalb des „Webpagetest.org“ – Tools gepushte Ressourcen akzeptiert werden. Wenn die Webapplikation am lokalen PC im Browser „Google Chrome“ aufgerufen wurde, wurde festgestellt, dass per „Server Push“ übergebene Ressourcen nicht vollständig den Client erreichen und neu angefragt und geladen wurden. Dies spricht dafür, dass der „Server Push“ des gewählten Servers im Browser „Google Chrome“ nicht vollständig funktioniert.

Man muss beachten, dass oben genannte Aussagen und Testergebnisse sich auf eine konkrete Testumgebung (Apache Server der Version 2.4.20 und drei Browser mit konkreten Versionen) beziehen. Mit jedem Release sowohl des Webservers als auch des Browsers können sich oben genannte Ergebnisse ändern.

Obwohl das offizielle Datum der Veröffentlichung des HTTP/2 – Protokolls schon über ein Jahr zurück liegt, ist zu beobachten, dass bisher nicht viele Tests und Experimente durchgeführt und öffentlich gemacht wurden. Zum aktuellen Stand dieses Protokolls kann man sagen, dass es sowohl server- als auch clientseitig nicht perfekt funktioniert und manche der erzielten Ergebnisse unerwartet waren und teilweise unerklärt geblieben sind. Deshalb konnten im Rahmen dieser Masterarbeit nicht alle am Anfang definierten Tests komplett durchgeführt werden, da manche Funktionen nicht korrekt implementiert waren. Außerdem unterscheidet sich das Verhalten des neuen Protokolls zwischen unterschiedlichen Browsern sehr.

Für die zukünftige Entwicklung des HTTP/2 – Protokolls ist es auf jeden Fall empfehlenswert, das neue Protokoll unter anderen Bedingungen (andere Server und unterschiedliche Browser) immer weiter zu testen.

Die Verfasserin dieser Masterarbeit schlägt vor, bei weiteren Untersuchungen zu optimalen Bedingungen für clientseitige Techniken zur Performance – Optimierung auf allgemeine Aussagen zu verzichten und stattdessen für die spezifische Umgebung individuelle Tests durchzuführen. Denn innerhalb dieser Masterarbeit konnte z.B. gezeigt werden, dass nicht zusammengefasste Ressourcen langsamer heruntergeladen werden, obwohl dies laut den allgemeinen Aussagen nicht zu erwarten gewesen wäre.

7. Literaturverzeichnis

Apache Software Foundation 2016a
Apache Software Foundation, Apache Module mod_http2. <https://httpd.apache.org/docs/2.4/mod/mod_http2.html> (24. September 2016).

Apache Software Foundation 2016b
Changes with Apache. <https://svn.apache.org/repos/asf/httpd/httpd/branches/2.4.x/CHANGES> (24. September 2016).

Apple 2016
Apple, Web Development Tools. 2016. <https://developer.apple.com/safari/tools/> (24. September 2016).

Behnke 2013
M. Behnke, Mime Types for Fonts and Media. 2013. <https://gist.github.com/localpcguy/6002288> (24. September 2016).

Belshe et al. 2015
M. Belshe/R. Peon/ M. Thomson, Hypertext Transfer Protocol Version 2 (HTTP/2). 2015. <https://tools.ietf.org/pdf/rfc7540.pdf> (24. September 2016).

CloudFlare 2016
CloudFlare, Tools for debugging, testing and using HTTP/2. 2016. <https://blog.cloudflare.com/tools-for-debugging-testing-and-using-http-2/> (24. September 2016).

DeNA Co., Ltd. 2015
DeNA Co., Ltd., H2O. 2015. <https://h2o.examp1e.net/> (24. September 2016).

Duran 2016
E. Duran, chromeHAR. 2016. <https://ericduran.github.io/chromeHAR/> (24. September 2016).

Duca/Glazkov 2016
N. Duca/D. Glazkov, Platform Success Model Explainer. 2016. <https://docs.google.com/document/d/1bYMyE6NdiAupuwl7pWQfB-vOZBPSsXCv57hljLDMV8E/edit#heading=h.116kzlmx4qct> (24. September 2016).

DistroWatch 2016
DistroWatch, DistroWatch Page Hit Ranking. 2016. <https://distrowatch.com/dwres.php?resource=popularity> (24. September 2016).

Equation Research 2011
Equation Research, What users want from mobile. 2011. <http://www.slideshare.net/RFONNIER/what-users-want-from-mobile-equation-research-july-2011> (24. September 2016).

Eissing 2015
S. Eissing, how to h2 in apache. 2016. <https://icing.github.io/mod_h2/howto.html> (24. September 2016).

Fractal 2016
Fractal, Gulpjs. 2016. <http://gulpjs.com/> (24. September 2016).

Fielding et al. 1999
R. Fielding/J. Gettys/J. Mogul/H. Frystyk/L. Masinter/P. Leach/T. Berners-Lee, Hypertext Transfer Protocol - HTTP/1.1. 1999. <https://www.ietf.org/rfc/rfc2616.txt> (24. September 2016).

Garbee 2016
J. Garbee, Understanding Resource Timing. 2016. <https://developers.google.com/web/tools/chrome-devtools/profile/network-performance/understanding-resource-timing> (24. September 2016).

Grigorik 2012
I. Grigorik, Deciphering the Critical Rendering Path. 2012. <http://calendar.perfplanet.com/2012/deciphering-the-critical-rendering-path/> (24. September 2016).

Grigorik 2013
I. Grigorik, High Performance Browser Networking. Sebastopol 2003.

Grigorik 2015
I. Grigorik, HTTP/2. A New Excerption from High Performance Browser Networking. Sebastopol, 2015.

Grigorik 2016a
I. Grigorik, Measuring the critical rendering path with Navigation Timing. 2016. <https://developers.google.com/web/fundamentals/performance/critical-rendering-path/measure-crp?hl=en> (24. September 2016).

Grigorik 2016b
I. Grigorik, Bildoptimierung. 2016. <https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/image-optimization> (24. September 2016).

Grigorik 2016c
I. Grigorik, Constructing the Object Model. 2016. <https://developers.google.com/web/fundamentals/performance/critical-rendering-path/constructing-the-object-model?hl=en> (24. September 2016).

Grigorik 2016d
I. Grigorik, Render-tree construction, layout, and paint. 2016. <https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction?hl=en> (24. September 2016).

Grigorik 2016e
I. Grigorik, Render blocking CSS. 2016. <https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-blocking-css?hl=en> (24. September 2016).

Grigorik 2016f
I. Grigorik, Adding interactivity with JavaScript. 2016. <https://developers.google.com/web/fundamentals/performance/critical-rendering-path/adding-interactivity-with-javascript?hl=en> (24. September 2016).

Grigorik 2016g
I. Grigorik, Critical rendering path. 2016. <https://developers.google.com/web/fundamentals/performance/critical-rendering-path/?hl=en> (24. September 2016).

Grigorik 2016h
I. Grigorik, Analyzing critical rendering path performance. 2016. <https://developers.google.com/web/fundamentals/performance/critical-rendering-path/analyzing-crp?hl=en> (24. September 2016).

Grigorik 2016i
I. Grigorik, Web font optimization. 2016. <https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/webfont-optimization?hl=en#optimizing-loading-and-rendering> (24. September 2016).

Grunt Development Team 2016
Grunt Development Team, Grunt. 2016. <http://gruntjs.com/> (24. September 2016).

Holtkamp 2001
H. Holtkamp, TCP/IP im Detail. 2001. <http://www.rvs.uni-bielefeld.de/~heiko/tcpip/tcpip_html_alt/kap_2_4.html> (24. September 2016).

Hemming 2016
T. Hemming, HTTP Archive Viewer. 2016. <https://chrome.google.com/webstore/detail/http-archive-viewer/ebbdbdmhegaoooipfnjikefdpeoaidml?hl=de> (24. September 2016).

Ishizawa 2015
M. Ishizawa, Understanding HTTP/2 prioritization. 2015. <https://speakerdeck.com/summerwind/2-prioritization> (24. September 2016).

Jayaprakash 2016
A. Jayaprakash, Are you ready for HTTP/2 Server Push. 2016?! <https://blogs.akamai.com/2016/04/are-you-ready-for-http2-server-push.html> (24. September 2016).

Krasnov 2016
V. Krasnov, Announcing Support for HTTP/2 Server Push. 2016. <https://blog.cloudflare.com/announcing-support-for-http-2-server-push-2/> (24. September 2016).

Kuhn/Raith 2013
D. Kuhn, M. Raith, Performante Webanwendungen. Client- und serverseitige Techniken zur Performance-Optimierung. Heidelberg 2013.

Lewis 2016
P. Lewis, Rendering performance. 2016. <https://developers.google.com/web/fundamentals/performance/rendering/?hl=en> (24. September 2016).

Lighttpd 2016
Lighttpd. 2016. <https://www.lighttpd.net/> (24. September 2016).

McLachlan 2013
P. McLachlan, On Mobile, Data URIs are 6x Slower than Source Linking (New Research). 2013. <http://www.mobify.com/blog/data-uris-are-slow-on-mobile/> (24. September 2016).

Medienmaster.de 2016
Medienmaster.de, Medienmaster.de. 2016. <http://www.medienmaster.de/> (24. September 2016).

Microsoft 2016
Microsoft, Analysieren des Netzwerkdatenverkehrs einer Webseite. 2016. <https://msdn.microsoft.com/de-de/library/dn255004(v=vs.85).aspx> (24. September 2016).

Mishunov 2015
D. Mishunov, Why Performance Matters, Part 1: The Perception Of Time. 2015. <https://www.smashingmagazine.com/2015/09/why-performance-matters-the-perception-of-time/> (24. September 2016).

Mifsud 2016
M. Mifsud, Real–world HTTP/2: 400gb of images per day. 2016. <https://99designs.de/tech-blog/blog/2016/07/14/real-world-http-2-400gb-of-images-per-day/> (24. September 2016).

Mozilla Developer Network and individual contributors 2016a
Mozilla Developer Network and individual contributors, <script>. 2016. <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script> (24. September 2016).

Mozilla Developer Network and individual contributors 2016b
Mozilla Developer Network and individual contributors, Navigation Timing API. 2016. <https://developer.mozilla.org/en-US/docs/Web/API/Navigation_timing_API> (24. September 2016).

Mozilla Developer Network and individual contributors 2016c
Mozilla Developer Network and individual contributors, PerformanceTiming. 2016. <https://developer.mozilla.org/en-US/docs/Web/API/PerformanceTiming> (24. September 2016).

Mozilla Developer Network and individual contributors 2016d
Mozilla Developer Network and individual contributors, Document.readyState. 2016. <https://developer.mozilla.org/en-US/docs/Web/API/Document/readyState> (24. September 2016).

Mozilla Developer Network and individual contributors 2016e
Mozilla Developer Network and individual contributors, DOMContentLoaded. 2016. <https://developer.mozilla.org/en-US/docs/Web/Events/DOMContentLoaded> (24. September 2016).

Mozilla Developer Network and individual contributors 2016f
Mozilla Developer Network and individual contributors, load. 2016. <https://developer.mozilla.org/en-US/docs/Web/Events/load> (24. September 2016).

Mozilla Developer Network and individual contributors 2016g
Mozilla Developer Network and individual contributors, What are browser developer tools. 2016. <https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_are_browser_developer_tools> (24. September 2016).

Mozilla Developer Network and individual contributors 2016h
Mozilla Developer Network and individual contributors, Using the Resource Timing API. 2016. <https://developer.mozilla.org/en-US/docs/Web/API/Resource_Timing_API/Using_the_Resource_Timing_API>

Mozilla Developer Network and individual contributors 2016i
Mozilla Developer Network and individual contributors, Network Monitor. 2016. <https://developer.mozilla.org/en-US/docs/Tools/Network_Monitor> (24. September 2016).

NGINX, Inc. 2015
NGINX, HTTP/2 for Web Application Developers. 2015. <https://assets.wp.nginx.com/wp-content/uploads/2015/09/NGINX_HTTP2_White_Paper_v4.pdf> (24. September 2016).

NGINX, Inc. 2016
NGINX, NGINX. 2016. <https://nginx.org/en/> (24. September 2016).
OpenSignal 2016a
OpenSignal, Telekom Netzabdeckungskarten. 2016. <http://opensignal.com/networks/deutschland/telekom-Berichterstattung> (24. September 2016).

OpenSignal 2016b
OpenSignal, Vodafone Netzabdeckungskarten. 2016. <http://opensignal.com/networks/deutschland/vodafone-Berichterstattung> (24. September 2016).

OpenSignal 2016c
OpenSignal, E-Plus Netzabdeckungskarten. 2016. <http://opensignal.com/networks/deutschland/e-plus-Berichterstattung> (24. September 2016).

OpenSignal 2016d
OpenSignal, O2 Netzabdeckungskarten. 2016. <http://opensignal.com/networks/deutschland/o2-Berichterstattung> (24. September 2016).

OPERA SOFTWARE ASA 2016
OPERA SOFTWARE ASA, Opera Dragonfly documentation. 2016. <http://www.opera.com/dragonfly/documentation/network/> (24. September 2016).

Peon/Ruellan 2015
R. Peon/H. Ruellan, HPACK: Header Compression for HTTP/2. 2015. <https://tools.ietf.org/pdf/rfc7541.pdf> (24. September 2016).

Red Hat, Inc. and others 2015
Red Hat, Inc. and others, Fedora. 2015. <https://getfedora.org/> (24. September 2016).

Rigor, Inc. 2016
Rigor, Inc., The Case for Fast Web Performance. 2016. <https://zoompf.com/business-case> (24. September 2016).

Ross 2016
D. Ross, HTTP/2 Server Push. 2016. <https://srd.wordpress.org/plugins/http2-server-push/> (24. September 2016).

RStudio 2016
RStudio, RStudio. 2016. <https://www.rstudio.com/home/> (24. September 2016).

Sexton 2015
P. Sexton, Critical rendering path. 2015. <https://varvy.com/pagespeed/critical-render-path.html> (24. September 2016).

Shaver 2015
J. Shaver. Decrypting TLS Browser Traffic With Wireshark – The Easy Way. 2016. <https://jimshaver.net/2015/02/11/decrypting-tls-browser-traffic-with-wireshark-the-easy-way/> (24. September 2016).

Souders 2010
S. Sounders, Browser Performance Wishlist. 2010. <http://www.stevesoud-ers.com/blog/2010/02/15/browser-performance-wishlist/> (24. September 2016).

StatCounter 2016
StatCounter, StatCounter Global Stats. 2016. <http://gs.statcounter.com/#desktop-browser_version_partially_combined-ww-monthly-201507-201607-ba> (24. September 2016).

Suse 2013
Suse, Administration Guide. 2013. <https://www.suse.com/documenta-tion/sles11/book_sle_admin/data/sec_apache2_configuration.html> (24. September 2016).

SUSE LLC 2015
SUSE LLC, openSUSE. 2015. <https://www.opensuse.org/> (24. September 2016).

The jQuery Foundation 2016a
The jQuery Foundation, Using jQuery Core. 2016. <https://learn.jquery.com/using-jquery-core/document-ready/> (24. September 2016).

The jQuery Foundation 2016b
The jQuery Foundation, Ready. 2016. <https://api.jquery.com/ready/> (24. September 2016).

The R Foundation 2016
The R Foundation, The R Project for Statistical Computing. 2016. <https://www.r-pro-ject.org/> (24. September 2016).

Thomson/Nottingham 2016
M. Thomson/M. Nottingham, Implementations. 2016. <https://github.com/http2/http2-spec/wiki/Implementations> (24. September 2016).

Tsujikawa 2016
T. Tsujikawa, Nghttp2: HTTP/2 C Library. 2016. <https://nghttp2.org/> (24. September 2016).

Viscomi et al. 2014
R. Viscomi/A. Davies/M. Duran, Using WebPagetest. Sebastopol, 2014.

WebPagetest 2016a
WebPagetest, WebPagetest. 2016. <https://www.webpagetest.org/> (24. September 2016).

WebPagetest 2016b
WebPagetest, WebPagetest Documentation. 2016.
<https://sites.google.com/a/webpagetest.org/docs/> (24. September 2016).

Wickham 2013
H. Wickham. Ggplot2. 2013. <http://ggplot2.org/> (24. September 2016).

Wikipedia 2016a
Wikipedia, Boxplot. 2016 <https://de.wikipedia.org/wiki/Boxplot> (24. September 2016).

Wikipedia 2016b
Wikipedia, Man-in-the-Middle-Angriff. 2016. <https://de.wikipedia.org/wiki/Man-in-the-Middle-Angriff> (24. September 2016).

Wikipedia 2016c
Wikipedia, HTTP pipelining. 2016. <https://en.wikipedia.org/wiki/HTTP_pipelining> (24. September 2016).

Wikipedia 2016d
Wikipedia, TCP congestion control. 2016. <https://en.wikipedia.org/wiki/TCP_congestion_control> (24. September 2016).
Wikipedia 2016e
Wikipedia, Data-URL. 2016. <https://de.wikipedia.org/wiki/Data-URL> (24. September 2016).

Wikipedia 2016f
Wikipedia, Hypertext Transfer Protocol. 2016. <https://de.wikipedia.org/wiki/Hypertext_Transfer_Protocol> (24. September 2016).

Wikipedia 2016g
Wikipedia, Application-Layer Protocol Negotiation. 2016. <https://de.wikipedia.org/wiki/Application-Layer_Protocol_Negotiation> (24. September 2016).

Wireshark Foundation 2016a
Wireshark Foundation, Wireshark. 2016. <https://www.wireshark.org/> (24. September 2016).

Wireshark Foundation 2016b
Wireshark Foundation, Hypertext Transfer Protocol version 2 (HTTP2). 2016.
<https://wiki.wireshark.org/HTTP2> (24. September 2016).

Wireshark Foundation 2016c
Wireshark Foundation , Display Filter Reference: HyperText Transfer Protocol 2. 2016. <https://www.wireshark.org/docs/dfref/h/http2.html> (24. September 2016).
W3C 2014a
W3C, HTML5. 2014. <https://www.w3.org/TR/html5/> (24. September 2016).

W3C 2014b
W3C, Scripting-HTML5. 2014. <https://www.w3.org/TR/html5/scripting-1.html> (24. September 2016).

W3C 2016a
W3C, W3C. 2016. <https://www.w3.org/> (24. September 2016).

W3C 2016b
W3C, Navigation Timing Level 2. 2016. <https://www.w3.org/TR/navigation-timing-2/> (24. September 2016).

W3C 2016c
W3C, Resource Timing Level 1. 2016. <https://www.w3.org/TR/resource-timing/> (24. September 2016).

8. Kurzfassung / Abstract

Kurzfassung

Heutzutage sind perfomante Webanwendungen, die schnell geladen werden und genauso schnell mit den Nutzern interagieren können, immer gefragter. Dabei spielt die Webperformance – Optimierung eine große Rolle. Im Mai 2015 ist die neue Version des HTTP – Protokolls (HTTP/2) erschienen, mithilfe deren Verwendung die Datenübertragung schneller und effizienter sein sollte. Hinsichtlich der technologischen Möglichkeiten des HTTP/2 – Protokolls kann der Austausch zwischen dem Client und dem Server deutlich beschleunigt werden. In der aktuellen Masterarbeit werden die Techniken zu Frontend – Optimierungstechniken evaluiert, die unter der Verwendung des neuen Protokolls gut geeignet sein können. Besonders wurde dabei auf die Funktion „Server Push" geachtet. Außerdem wird dargestellt, welche notwendigen Maßnahmen die Webentwickler unternehmen müssen, um das neue Protokoll benutzen zu können. Unter anderem wird evaluiert, wie gut aktuelle Implementierungen des neuen Protokolls und der dazugehörigen Funktionen funktionieren und wie sich die Ladezeit zwischen dem aktuell meist verwendeten Protokoll HTTP/1.1- und dem HTTP/2 – Protokoll unterscheidet.

Abstract

Nowadays, dynamic web-applications are growing in popularity because of their quality of fast loading and their immediate interaction with the user. Web performance optimization plays an important role in this process. The newest version of the HTTP – Protocol, HTTP/2, was published in May 2015 which should increase the speed of data transmission and make it more efficient. With regard to the technological possibilities of the HTTP/2 – Protocol, the transfer between server and client can be accelerated. This master thesis will evaluate the technique of front end optimization which might work well with the new protocol. A particular focus of the thesis will be the function "Server Push". Furthermore, necessary measures of web development will be outlined to show the optimal usage of the protocol. Amongst other things, it will be explained how current implementations of the protocol with their functions work and it what ways the page load time of the current HTTP/1.1 – Protocol and the new published HTTP/2 – Protocol differ from each other.

9. Anhang

9.1. Upgrade HTTP/2

Erster Fall: Es wird eine verschlüsselte Verbindung angefragt

```
> curl -v https://im-im2.hdm-stuttgart.de/
*   Trying 141.62.110.42...
* Connected to im-im2.hdm-stuttgart.de (141.62.110.42) port 443 (#0)
* ALPN, offering h2
* ALPN, offering http/1.1
* Cipher selection: ALL:!EXPORT:!EXPORT40:!EXPORT56:!aNULL:!LOW:!RC4:@STRENGTH
* successfully set certificate verify locations:
*   CAfile: D:\CURL\ca-bundle.crt
  CApath: none
* TLSv1.2 (OUT), TLS header, Certificate Status (22):
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Server hello (2):
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (IN), TLS handshake, Server key exchange (12):
* TLSv1.2 (IN), TLS handshake, Server finished (14):
* TLSv1.2 (OUT), TLS handshake, Client key exchange (16):
* TLSv1.2 (OUT), TLS change cipher, Client hello (1):
* TLSv1.2 (OUT), TLS handshake, Finished (20):
* TLSv1.2 (IN), TLS change cipher, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Finished (20):
* SSL connection using TLSv1.2 / ECDHE-RSA-AES128-GCM-SHA256
* ALPN, server accepted to use h2
* Server certificate:
*  subject: C=DE; ST=Baden-Wuerttemberg; L=Stuttgart; O=Hochschule der Medien Stutt-
gart; OU=InteraktiveMedien; CN=im-im2.hdm-stuttgart.de
*  start date: Jun  8 08:56:10 2016 GMT
*  expire date: Jul  9 23:59:00 2019 GMT
*  subjectAltName: host "im-im2.hdm-stuttgart.de" matched cert's "im-im2.hdm-stuttgart.de"
*  issuer: C=DE; O=DFN-Verein; OU=DFN-PKI; CN=DFN-CA Global
*  SSL certificate verify ok.
* Using HTTP2, server supports multi-use
* Connection state changed (HTTP/2 confirmed)
* TCP_NODELAY set
* Copying HTTP/2 data in stream buffer to connection buffer after upgrade: len=0
```

```
* Using Stream ID: 1 (easy handle 0x400560)
> GET / HTTP/1.1
> Host: im-im2.hdm-stuttgart.de
> User-Agent: curl/7.48.0
> Accept: */*
>
* Connection state changed (MAX_CONCURRENT_STREAMS updated)!
< HTTP/2.0 200
< date:Fri, 26 Aug 2016 12:48:24 GMT
< server:Apache
< last-modified:Tue, 02 Aug 2016 07:39:47 GMT
< etag:"4b1b-53911d22dd6c0"
< accept-ranges:bytes
< content-length:19227
< vary:Accept-Encoding
< cache-control:max-age=3600
< expires:Fri, 26 Aug 2016 13:48:24 GMT
< content-type:text/html
```

Zweiter Fall: Es wird eine unverschlüsselte Verbindung angefragt

```
> curl -v http://im-im2.hdm-stuttgart.de/
*   Trying 141.62.110.42...
* Connected to im-im2.hdm-stuttgart.de (141.62.110.42) port 80 (#0)
> GET / HTTP/1.1
> Host: im-im2.hdm-stuttgart.de
> User-Agent: curl/7.48.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Fri, 26 Aug 2016 12:47:31 GMT
< Server: Apache
< Upgrade: h2
< Connection: Upgrade
< Last-Modified: Tue, 02 Aug 2016 07:39:47 GMT
< ETag: "4b1b-53911d22dd6c0"
< Accept-Ranges: bytes
< Content-Length: 19227
< Vary: Accept-Encoding
```

< Cache-Control: max-age=3600
< Expires: Fri, 26 Aug 2016 13:47:31 GMT
< Content-Type: text/html

9.2. Browserstatistik

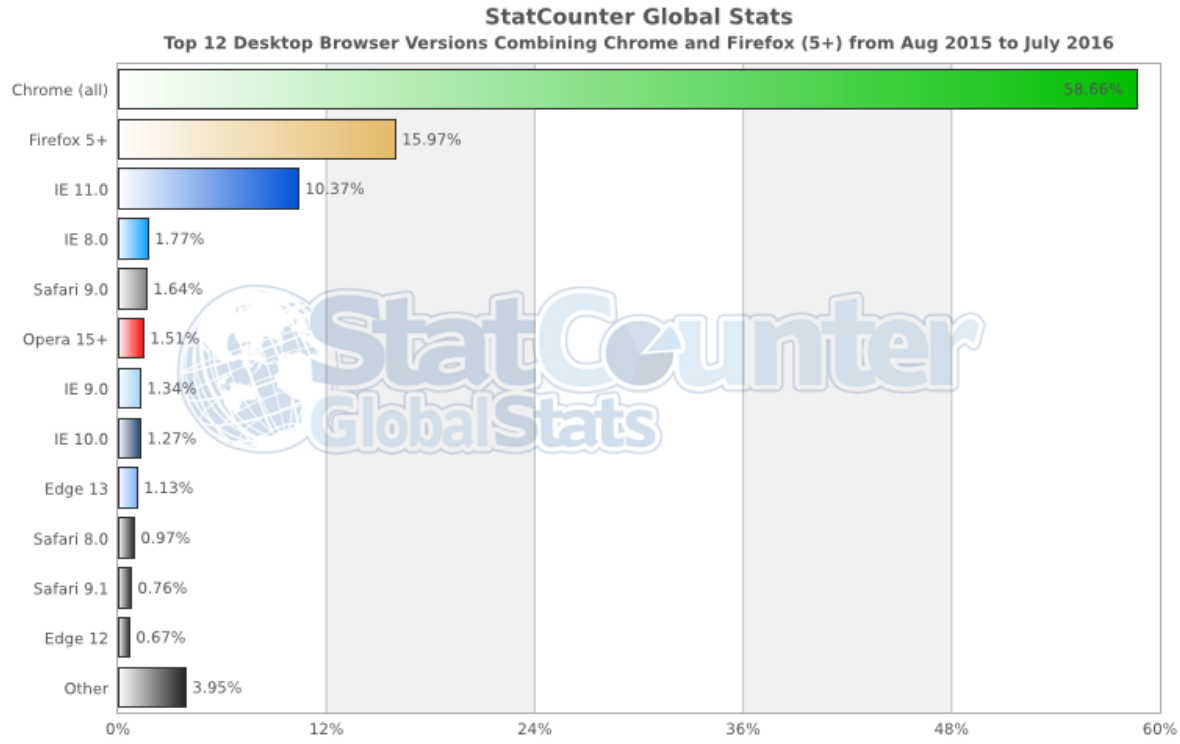

Abb. 60: Vergleich der am häufigsten eingesetzten Webbrowsern. Die Daten wurden zwischen Juli 2015 und Juli 2016 gemessen (<http://gs.statcounter.com/#desktop-browser_version_partially_combined-ww-monthly-201507-201607-ba>).